

DevOps in Python

Infrastructure as Python

Moshe Zadka

Apress®

DevOps in Python

Infrastructure as Python

Moshe Zadka

Apress®

DevOps in Python: Infrastructure as Python

Moshe Zadka
Belmont, CA, USA

ISBN-13 (pbk): 978-1-4842-4432-6 ISBN-13 (electronic): 978-1-4842-4433-3
https://doi.org/10.1007/978-1-4842-4433-3

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Celestin Suresh John
Development Editor: James Markham
Coordinating Editor: Divya Modi

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit http://www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-4432-6. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

To A and N, my favorite two projects — even when they need immediate maintenance at 4 a.m.

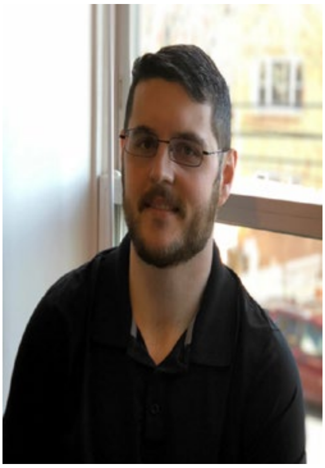
Table of Contents

About the Author



Moshe Zadka has been part of the open source community since 1995 and has been involved with DevOps since before the term became mainstream. One of two collaborators in the Facebook bootcamp Python class, he made his first core Python contributions in 1998, and is a founding member of the Twisted open source project. He has also given tutorials and talks at several recent PyCon conferences and is a co-author of *Expert Twisted* (Apress, 2019).

About the Technical Reviewer



Paul Ganssle is a software developer at Bloomberg and frequent contributor to open source projects. Among other projects, he maintains the Python libraries dateutil and setuptools. He holds a PhD in Physical Chemistry from the University of California, Berkeley; and a BS in Chemistry from the University of Massachusetts, Amherst.

Acknowledgments

Thanks to my wife, Jennifer Zadka, without whose support I could not have done it.

Thanks to my parents, Yaacov and Pnina Zadka, who taught me how to learn.

Thanks to my advisor, Yael Karshon, for teaching me how to write.

Thanks to Mahmoud Hashemi for inspiration and encouragement.

Thanks to Mark Williams for always being there for me.

Thanks to Glyph Lefkowitz for teaching me things about Python, about programming, and about being a good person.

Introduction

Python was started as a language to automate an operating system: the Amoeba. Since it had an API, not just textual files representations, a typical UNIX shell would be ill suited. The Amoeba OS is now a relic. However, Python continues to be a useful tool for automation of operations – the heart of typical DevOps work. It is easy to learn and easy to write readable code in – a necessity, when a critical part of the work is responding to a 4 a.m. alert, and modifying some misbehaving program. It has powerful bindings to C and C++, the universal languages of operating systems – and yet is natively memory safe, leading to few crashes at the automation layer.

Finally, although not true when it was originally created, Python is one of the most popular languages. This means that it is relatively easy to hire people with Python experience, and easy to get training materials and courses for people who need to learn on the job.

This book will guide you through how to take advantage of Python to automate operations.

What to Expect in the Book

There are many sources that teach Python, the language: books, tutorials, and even free videos online. Basic familiarity with the language will be assumed here. However, for the typical SRE/DevOps person, there are a lot of aspects of Python that few sources cover, except for primary documentation and various blogs. We cover those early on in the book.

The first step in using Python is not writing a "hello world" program. The first step is installing it. There, already, we are faced with various choices, with various trade-offs between them. We will cover using the preinstalled version of Python, using ready-made, third-party prebuilt packages, installing Python from sources, and other alternatives. One of Python's primary strengths, which any program slightly longer than "hello world" will take advantage of, is its rich third-party library ecosystem. We will cover

how to use these packages, how to develop a workflow around using specific versions of packages and when to upgrade, and what tools are used to manage that. We will also cover packaging our own code, whether for open source distribution or for internal dissemination.

Finally, Python was built for exploration. Coming from the Lisp tradition of the REPL (Read-Eval-Print Loop), using Python interactively is a primary way to explore the language, the libraries, and even the world around us. Whether the world is made of planets and atoms, or virtual machines and containers, the same tools can be used to explore it. Being a DevOps-oriented book, we cover it more from the perspective of exploring a world of virtual machines, services, and containers.

CHAPTER 1

Installing Python

Before we start using Python, we need to install it. Some operating systems, like Mac OS X and some Linux variants, have Python preinstalled. Those versions of Python, colloquially called "system Python," often make poor defaults for people who want to develop in Python.

For one thing, the version of Python installed is often behind latest practices. For another, system integrators will often patch Python in ways that can lead to surprises later. For example, Debian-based Python is often missing modules like `venv` and `ensurepip`. Mac OS X Python links against a Mac shim around its native SSL library. What those things means, especially when starting out and consuming FAQs and web resources, is that it is better to install Python from scratch.

We cover a few ways to do so and the pros and cons of each.

1.1 OS Packages

For some of the more popular operating systems, volunteers have built ready-to-install packages.

The most famous of these is the "deadsnakes" PPA (Personal Package Archives). The "dead" in the name refers to the fact that those packages are already built – with the metaphor that sources are "alive." Those packages are built for Ubuntu and will usually support all the versions of Ubuntu that are still supported upstream. Getting those packages is done simply:

```
$ sudo add-apt-repository ppa:deadsnakes/ppa
$ sudo apt update
```


M. Zadka, *DevOps in Python*, https://doi.org/10.1007/978-1-4842-4433-3_1

On Mac OS, the `homebrew` third-party package manager will have Python packages that are up to date. An introduction to Homebrew is beyond the scope of this book. Since Homebrew is a rolling release, the version of Python will get upgraded from time to time. While this means that it is a useful way to get an up-to-date Python, it makes for a poor target for reliably distributing tools.

It is also a choice with some downsides for doing day-to-day development. Since it upgrades quickly after a new Python releases, this means development environments can break quickly and without warning. It also means that sometimes code can stop working: even if you are careful to watch upcoming Pythons for breaking changes, not all packages will. Homebrew Python is a good fit when needing a well-built, up-to-date Python interpreter for a one-off task. Writing a quick script to analyze data, or automate some APIs, is a good use of Homebrew Python.

Finally, for Windows, it is possible to download an installer from Python.org for any version of Python.

1.2 Using Pyenv

Pyenv tend to be the highest Return on Investment for installing Python for local development. The initial setup does have some subtleties. However, it allows installing as many Python versions side by side as are needed. It allows managing how one will be accessed: on either per-user default or a per-directory default.

Installing pyenv itself depends on the operating system. On a Mac OS X, the easiest way is to install it via Homebrew. Note that in this case, pyenv itself might need to be upgraded to install new versions of Python.

On a UNIX-based operating system, such as Linux or FreeBSD, the easiest way to install pyenv is by using the `curl|bash` command:

```
$ PROJECT=https://github.com/pyenv/pyenv-installer \
  PATH=raw/master/bin/pyenv-installer \
  curl -L $PROJECT/PATH | bash
```

Of course, this comes with its own security issues and could be replaced with a two-step process:

```
$ git clone https://github.com/pyenv/pyenv-installer
$ cd pyenv-installer
$ bash pyenv-installer
```

where one can inspect the shell script before running, or even use `git checkout` to pin to a specific revision.

Sadly, pyenv does not work on Windows.

After installing pyenv, it is useful to integrate it with the running shell. We do this by adding to the shell initialization file (for example, `.bash_profile`):

```
export PATH="~/.pyenv/bin:$PATH"
eval "$(pyenv init -)"
eval "$(pyenv virtualenv-init -)"
```

This will allow pyenv to properly intercept all needed commands.

Pyenv separates the notion of *installed* interpreters from *available* interpreters. In order to install a version,

```
$ pyenv install <version>
```

For CPython, `<version>` is just the version number, such as `3.6.6` or `3.7.0rc1`.

An *installed* version is distinct from an available version. Versions can be available either "globally" (for a user) by using

```
$ pyenv global 3.7.0
```

or locally by using

```
$ pyenv local 3.7.0
```

Local means they will be available in a given directory. This is done by putting a file `python-version.txt` in this directory. This is important for version-controlled repositories, but there are a few different strategies to manage those. One is to add this file to the "ignored" list. This is useful for heterogenous teams of open source projects. Another is to check this file in, so that the same version of Python is used in this repository.

Note that *pyenv*, since it is designed to install versions of Python side by side, has no concept of "upgrading" Python. In order to use a newer Python, it needs to be installed with *pyenv* and then set as the default.

By default, *pyenv* installs non-optimized versions of Python. If optimized versions are needed,

```
env PYTHON_CONFIGURE_OPTS="--enable-shared
                           --enable-optimizations
                           --with-computed-gotos
                           --with-lto
                           --enable-ipv6" pyenv install
```

will build a version that is pretty similar to binary versions from `python.org`.

1.3 Building Python from Source

The main challenge in building Python from source is that, in some sense, it is too forgiving. It is all too easy to build it with one of the built-in modules disabled because its dependency was not detected. This is why it is important to know what dependencies are that fragile, and how to make sure a local installation is good.

The first fragile dependency is `ssl`. It is disabled by default and must be enabled in `Modules/Setup.dist`. Carefully follow the instructions there about the location of the OpenSSL library. If you have installed OpenSSL via system packages, it will usually be in `/usr/`. If you have installed it from source, it will usually be in `/usr/local`.

The most important thing is to know how to test for it. When Python is done building, run `./python.exe -c 'import _ssl'`. That `.exe` is not a mistake – this is how the build process calls the just-built executable, which is renamed to `python` during installation. If this succeeds, the `ssl` module was built correctly.

Another extension that can fail to build is `sqlite`. Since it is a built-in, a lot of third-party packages depend on it, even if you are not using it yourself. This means a Python installation without the `sqlite` module is pretty broken. Test by running `./python.exe -c 'import sqlite3'`.

In a Debian-based system (such as Ubuntu), `libsqlite3-dev` is required for this to succeed. In a Red Hat-based system (such as Fedora or CentOS), `libsqlite3-dev` is required for this to succeed.

Next, check for `_ctypes` with `./python.exe -c 'import _ctypes'`. If this fails, it is likely that the `libffi` headers are not installed.

Finally, remember to run the built-in regression test suite after building from source. This is there to ensure that there have been no silly mistakes while building the package.

1.4 PyPy

The "usual" implementation of Python is sometimes known as "CPython," to distinguish it from the language proper. The most popular alternative implementation is PyPy. PyPy is a Python-based JIT implementation of Python in Python. Because it has a dynamic JIT (Just in Time) compilation to assembly, it can sometimes achieve phenomenal speed-ups (3x or even 10x) over regular Python.

There are sometimes challenges in using PyPy. Many tools and packages are tested only with CPython. However, sometimes spending the effort to check if PyPy is compatible with the environment is worth it if performance matters.

There are a few subtleties in installing Python from source. While it is theoretically possible to "translate" using CPython, in practice the optimizations *in* PyPy mean that translating using PyPy works on more reasonable machines. Even when installing from source, it is better to first install a binary version to bootstrap.

The bootstrapping version should be PyPy, not PyPy3. PyPy is written in the Python 2 dialect. It is one of the only cases where worrying about the deprecation is not relevant, since PyPy is a Python 2 dialect interpreter. PyPy3 is the Python 3 dialect implementation, which is usually better to use in production as most packages are slowly dropping support for Python 2.

The latest PyPy3 supports 3.5 features of Python, as well as f-strings. However, the latest async features, added in Python 3.6, do not work.

1.5 Anaconda

The closest to a "system Python" that is still reasonable for use as a development platform is the Anaconda Python. Anaconda is a so-called "meta-distribution." It is, in essence, an operating system on top of the operating system. Anaconda has its grounding in the scientific computing community, and so its Python comes with easy-to-install modules for many scientific applications. Many of these modules are nontrivial to install from PyPI, requiring a complicated build environment.

It is possible to install multiple Anaconda environments on the same machine. This is handy when needing different Python versions or different versions of PyPI modules.

In order to bootstrap Anaconda, we can use the `bash` installer, available from `https://conda.io/miniconda.html`. The installer will also modify `~/.bash_profile` to add the path to `conda`, the installer.

Conda environments are created using conda create --name <name>, and activated using source conda activate <name>. There is no easy way to use unactivated environments. It is possible to create a conda environment while installing packages in it: conda create --name some-name python. We can specify the version using = - conda create --name some-name python=3.5. It is also possible to install more packages into a conda environment, using conda install package[=version], after the environment has been activated. Conda has a lot of pre-built Python packages, especially ones that are nontrivial to build locally. This makes it a good choice if those packages are important to your use case.

1.6 Summary

Running Python program requires an interpreter installed on the system. Depending on the operating system, and needed versions, there are several different ways to install Python. Using the system Python is a problematic option. On Mac and UNIX systems, using pyenv is almost always the preferred option. On Windows, using the prepackaged installers from python.org is often a good idea.

CHAPTER 2

Packaging

Much of dealing with Python in the real world is dealing with third-party packages. For a long time, the situation was not good. Things have improved dramatically, however. It is important to understand which "best practices" are antiquated rituals, which ones are based on faulty assumptions but have some merit, and which are actually good ideas.

When dealing with packaging, there are two ways to interact. One is to be a "consumer," wanting to use the functionality from a package. Another is to be the "producer," publishing a package. These describe, usually, different development tasks, not different people.

It is important to have a solid understanding of the "consumer" side of packages before moving to "producing." If the goal of a package publisher is to be useful to the package user, it is crucial to imagine the "last mile" before starting to write a single line of code.

2.1 Pip

The basic packaging tool for Python is `pip`. By default, installations of Python do not come with `pip`. This allows `pip` to move faster than core Python – and work with alternative Python implementations, like PyPy. However, they do come with the useful `ensurepip` module. This allows getting pip via `python -m ensurepip`. This is usually the easiest way to bootstrap `pip`.

Some Python installations, especially system ones, disable `ensurepip`. When lacking `ensurepip`, there is a way of manually getting it: `get-pip.py`. This is a downloadable single file that, when executed, will unpack `pip`.

Luckily, `pip` is the only package that needs these weird gyrations to install. All other packages can, and should, be installed using `pip`. This includes upgrading `pip` itself, which can be done with `pip install --upgrade pip`.

M. Zadka, *DevOps in Python*, https://doi.org/10.1007/978-1-4842-4433-3_2

Depending on how Python was installed, its “real environment” might or might not be modifiable by our user. Many instructions in various README files and blogs might encourage doing *sudo pip install*. This is almost always the wrong thing to do: it will install the packages in the global environment.

It is almost always better to install in virtual environments – those will be covered later. As a temporary measure, perhaps to install things needed to create a virtual environment, we can install to our *user* area. This is done with `pip install --user`.

The `pip install` command will download and install all dependencies. However, it can fail to downgrade incompatible packages. It is always possible to install explicit versions: `pip install package-name==<version>` will install this precise version. This is also a good way to get explicitly non-general-availability packages, such as release candidates, beta, or similar, for local testing.

If `wheel` is installed, `pip` will build, and usually cache, wheels for packages. This is especially useful when dealing with a high virtual environment churn, since installing a cached wheel is a fast operation. This is also especially useful when dealing with so-called “native,” or “binary,” packages – those that need to be compiled with a C compiler. A wheel cache will eliminate the need to build it again.

`pip` does allow uninstalling, with `pip uninstall`. This command, by default, requires manual confirmation. Except for exotic circumstances, this command is not used. If an unintended package has snuck in, the usual response is to destroy the environment and rebuild it. For similar reasons, `pip install --ugprade` is not often needed: the common response is to destroy and re-create the environment. There is one situation where it is a good idea: `pip install --upgrade pip`. This is the best way to get a new version of `pip` with bug fixes and new features.

`pip install` supports a “requirements file,” `pip install --requirements` or `pip install -r`. A requirements file simply has one package per line. This is no different than specifying packages on the command line. However, requirements files often specify “strict dependencies.” A requirements file can be *generated* from an environment with `pip freeze`. The usual way to get a requirements file is, in a virtual environment, to do the following:

```
$ pip install -e .
$ pip freeze > requirements.txt
```

This means the requirements file will have the current package, and all of its recursive dependencies, with strict versions.

2.2 Virtual Environments

Virtual environments are often misunderstood, because the concept of "environments" is not clear. A Python environment refers to the root of the Python installation. The reason it is important is because of the subdirectory `lib/site-packages`. The `lib/site-packages` directory is where third-party packages are installed. In modern times, they are often installed by `pip`. While there used to be other tools to do it, even bootstrapping `pip` and `virtualenv` can be done with `pip`, let alone day-to-day package management.

The only common alternative to `pip` is system packages, where a system Python is concerned. In the case of an Anaconda environment, some packages might be installed as part of Anaconda. In fact, this is one of the big benefits of Anaconda: many Python packages are custom built, especially those which are nontrivial to build.

A "real" environment is one that is based on the Python installation. This means that to get a new real environment, we must reinstall (and often rebuild) Python. This is sometimes an expensive proposition. For example, `tox` will rebuild an environment from scratch if any parameters are different. For this reason, *virtual* environments exist.

A virtual environment copies the minimum necessary out of the real environment to mislead Python into thinking that it has a new root. The exact details are not important, but what is important is that this is a simple command that just copies files around (and sometimes uses symbolic links).

There are two ways to use virtual environments: activated and unactivated. In order to use an unactivated virtual environment, which is most common in scripts and automated procedures, we explicitly call Python from the virtual environment.

This means that if we created a virtual environment in `/home/name/venvs/my-special-env`, we can call `/home/name/venvs/my-special-env/bin/python` to work inside this environment. For example, `/home/name/venvs/my-special-env/bin/python -m pip` will run `pip` but install in the virtual environment. Note that for entry-point-based scripts, they will be installed alongside Python, so we can run `/home/name/venvs/my-special-env/bin/pip` to install packages in the virtual environment.

The other way to use a virtual environment is to "activate" it. Activating a virtual environment in a bash-like shell means *sourcing* its activate script:

```
$ source /home/name/venvs/my-special-env/bin/activate
```

The sourcing sets a few environment variables, only one of which is actually important. The important variable is `PATH`, which gets prefixed by `/home/name/venvs/my-special-env/bin`. This means that commands like `python` or `pip` will be found there first. There are two cosmetic variables that get set: `VIRTUAL_ENV` will point to the root of the environment. This is useful in management scripts that want to be aware of virtual environments.

`PS1` will get prefixed with `(my-special-env),` which is useful for a visual indication of the virtual environment while working interactively in the console.

In general, it is a good practice to only install third-party packages inside a virtual environment. Combined with the fact that virtual environments are "cheap," this means that if one gets into a bad state, it is easy to just remove the whole directory and start from scratch. For example, imagine a bad package install that causes Python startup to fail. Even running `pip uninstall` is impossible, since `pip` fails on startup. However, the "cheapness" means we can remove the whole virtual environment and re-create it with a good set of packages.

Modern practice, in fact, is moving more and more toward treating virtual environments as semi-immutable: after creating them, there is a single stage of "install all required packages." Instead of modifying it if an upgrade is required, we destroy the environment, re-create, and reinstall.

There are two ways to create virtual environments. One way is portable between Python 2 and Python 3 – `virtualenv`. This needs to be bootstrapped in some way, since Python does not come with `virtualenv` preinstalled. There are several ways to accomplish this. If Python was installed using a packaging system, such as a system packager, Anaconda, or Homebrew, then often the same system will have packaged `virtualenv`. If Python is installed using `pyenv`, in a user directory, sometimes just using `pip install` directly into the "original environment" is a good option, even though it is an exception to the "only install into virtual environments." Finally, this is one of the cases `pip install --user` might be a good idea: this will install the package into the special "user area." Note that this means that sometimes it will not be in `$PATH`, and the best way to run it will be using `python-m virtualenv.`

If no portability is needed, `venv` is a Python 3-only way of creating a virtual environment. It is accessed as `python -m venv`, as there is no dedicated entry point. This solves the "bootstrapping" problem of how to install `virtualenv`, especially when using a nonsystem Python.

Whichever command is used to create the virtual environment, it will create the directory for the environment. It is best if this directory does not exist before that. A best practice is to remove it before creating the environment. There are also options about how to create the environment: which interpreter to use and what initial packages to install. For example, sometimes it is beneficial to skip `pip` installation entirely. We can then bootstrap `pip` in the virtual environment by using `get-pip.py`. This is a way to avoid a bad version of `pip` installed in the real environment – since if it is bad enough, it cannot even be used to upgrade `pip`.

2.3 Setup and Wheels

The term “third party” (as in “third-party packages”) refers to a someone other than the Python core developers (“first party”) or the local developers (“second party”). We have covered how to install “first-party” packages in the installation section. We used `pip` and `virtualenv` to install “third-party” packages. It is time to finally turn our attention to the missing link: local development and installing local packages, or “second-party” packages.

This is an area seeing a lot of new additions, like `pyproject.toml` and `flit`. However, it is important to understand the classic way of doing things. For one, it takes a while for new best practices to settle in. For another, existing practices are based on `setup.py`, and so this way will continue to be the main way for a while – possibly even for the foreseeable future.

The `setup.py` file describes, in code, our “distribution.” Note that “distribution” is distinct from “package.” A package is a directory with (usually) `__init__.py` that Python can import. A distribution can contain several packages or even none! However, it is a good idea to keep a 1-1-1 relationship: one distribution, one package, named the same.

Usually, `setup.py` will begin by importing `setuptools` or `distutils`. While `distutils` is built-in, `setuptools` is not. However, it is almost always installed first in a virtual environment, due to its sheer popularity. Distutils is not recommended: it has not been updated for a long time. Notice that `setup.py` cannot meaningfully, explicitly declare it needs `setuptools` nor explicitly request a specific version: by the time it is read, it will have already tried to import `setuptools`. This non-declarativeness is part of the motivation for packaging alternatives.

The absolutely minimal `setup.py` that will work is the following:

```
import setuptools
setuptools.setup(
    packages=setuptools.find_packages(),
)
```

The official documentation calls a lot of other fields "required" for some reason, though a package will be built even if those are missing. For some, this *will* lead to ugly defaults, such as the package name being `UNKNOWN`.

A lot of those fields, of course, are good to have. But this skeletal `setup.py` is enough to create a distribution with the local Python packages in the directory.

Now, granted, almost always, there will be other fields to add. It is definitely the case that other fields will need to be added if this package is to be uploadable to a packaging index, even if it is a private index.

It is a great idea to add at least a "name" field. This will give the distribution a name. As mentioned earlier, it is almost always a good idea to name it after the single top-level package in the distribution.

A typical source hierarchy, then, will look like this:

```
setup.py
    import setuptools
    setuptools.setup(
        name='my_special_package',
        packages=setuptools.find_packages(),
    )
my_special_package/
    __init__.py
    another_module.py
    tests/
        test_another_module.py
```

Another field that is almost always a good idea is a version. Versioning software is, as always, hard. Even a running number, though, is a good way to answer the perennial question: "Is this running a newer or older version?"

There are some tools to help with managing the version number, especially assuming we want to have it also available to Python during runtime. Especially if doing Calendar Versioning, `incremental` is a powerful package to automate some of the

tedium. `bumpversion` is a useful tool, especially when choosing semantic versioning. Finally, `versioneer` supports easy integration with the git version control system, so that a tag is all that needs to be done for release.

Another popular field in `setup.py`, which is not marked “required” in the documentation but is present on almost every package, is `install_requires`. This is how we mark other distributions that our code uses. It is a good practice to put “loose” dependencies in `setup.py`. This is in contrast to exact dependencies, which specify a specific version. A loose dependency looks like `Twisted>=17.5` – specifying a minimum version but no maximum. Exact dependencies, like `Twisted==18.1`, are usually a bad idea in `setup.py`. They should only be used in extreme cases: for example, when using significant chunks of a package’s *private* API.

Finally, it is a good idea to give `find_packages` a whitelist of what to include, in order to avoid spurious files. For example,

```
setuptools.find_packages(include=["my_package*"])
```

Once we have `setup.py` and some Python code, we want to make it into a distribution. There are several formats a distribution can take, but the one we will cover here is the *wheel*. If `my-directory` is the one that has `setup.py`, running `pip wheel my-directory`, will produce a wheel, as well as the wheels of all of its recursive dependencies.

The default is to put the wheels in the current directory, which is seldom the desired behavior. Using `--wheel-dir<output-directory>` will put the wheel in the directory – as well as the wheels of any distribution it depends on.

There are several things we can do with the wheel, but it is important to note that one thing we can do is `pip install <wheel file>`. If we add `pip install <wheel file> --wheel-dir <output directory>`, then `pip` will use the wheels in the directory and will not go out to PyPI. This is useful for reproducible installs, or support for air-gapped modes.

2.4 Tox

Tox is a tool to automatically manage virtual environments, usually for tests and builds. It is used to make sure that those run in well-defined environments and is smart about caching them in order to reduce churn. True to its roots as a test-running tool, Tox is configured in terms of *test environments*.

It uses a unique ini-based configuration format. This can make writing configurations difficult, since remembering the subtleties of the file format can be hard. However, in return, there is a lot of power that, while being hard to tap, can certainly help in configuring tests and build runs that are clear and concise.

One thing that Tox does lack is a notion of *dependencies* between build steps. This means that those are usually managed from the outside, by running specific test runs after others and sharing artifacts in a somewhat ad hoc manner.

A Tox *environment* corresponds, more or less, to a section in the configuration file.

```
[testenv:some-name]
.
.
.
```

Note that if the name contains `pyNM` (for example, `py36`), then Tox will default to using CPython `N.M` (3.6, in this case) as the Python for that test environment. If the name contains `pypyNM`, Tox will default to using PyPy `N.M` for that version – where these stand for "version of CPython compatibility," not PyPy's own versioning scheme.

If the name does not include `pyNM` or `pypyNM`, or if there is a need to override the default, a `basepython` field in the section can be used to indicate a specific Python version. By default, Tox will look for these Pythons to be available in the path. However, if the plug-in `tox-pyenv` is installed, Tox will query `pyenv` if it cannot find the right Python on the path.

As examples, we will analyze one simple Tox file and one more complicated one.

```
[tox]
envlist = py36,pypy3.5,py36-flake8
```

The `tox` section is a global configuration. In this example, the only global configuration we have is the list of environments.

```
[testenv:py36-flake8]
```

This section configures the `py36-flake8` test environment.

```
deps =
    flake8
```

The deps subsection details which packages should be installed in the test environment's virtual environment. Here we chose to specify flake8 with a loose dependency. Another option is to specify a strict dependency (e.g., flake8==1.0.0.). This helps with reproducible test runs. We could also specify -r <requirements file> and manage the requirements separately. This is useful if we have other tooling that takes the requirements file.

```
commands =
    flake8 useful
```

In this case, the only command is to run flake8 on the directory useful. By default, a Tox test run will succeed if all commands return a successful status code. As something designed to run from command lines, flake8 respects this convention and will only exit with a successful status code if there were no problems detected with the code.

```
[testenv]
```

The other two environments, lacking specific configuration, will fall back to the generic environment. Note that because of their names, they will use two different interpreters: CPython 3.6 and PyPy with Python 3.5 compatibility.

```
deps =
    pytest
```

In this environment, we install the pytest runner. Note that in this way, our tox.ini documents the assumptions on the tools needed to run the tests. For example, if our tests used Hypothesis or PyHamcrest, this is where we would document it.

```
commands =
    pytest useful
```

Again, the command run is simple. Note, again, that pytest respects the convention and will only exit successfully if there were no test failures.

As a more realistic example, we turn to the tox.ini of ncolony:

```
[tox]
envlist = {py36,py27,pypy}-{unit,func},py27-lint,py27-wheel,docs
toxworkdir = {toxinidir}/build/.tox
```

We have more environments. Note that we can use the `{}` syntax to create a matrix of environments. This means that `{py36,py27,pypy}-{unit,func}` creates `3*2=6` environments. Note that if we had a dependency that made a “big jump” (for example, Django 1 and 2), and we wanted to test against both, we could have made `{py36,py27, pypy}-{unit,func}-{django1,django2}`, for a total of `3*2*2=12` environments. Notice the numbers for a matrix test like this climb up fast – and when using an automated test environment, it means things would either take longer or need higher parallelism.

This is a normal trade-off between comprehensiveness of testing and resource use. There is no magical solution other than to carefully consider how many variations to officially support.

```
[testenv]
```

Instead of having a `testenv` per variant, we choose to use one test environment but special case the variants by matching. This is a fairly efficient way to create many variants of test environments.

```
deps =
    {py36,py27,pypy}-unit: coverage
    {py27,pypy}-lint: pylint==1.8.1
    {py27,pypy}-lint: flake8
    {py27,pypy}-lint: incremental
    {py36,py27,pypy}-{func,unit}: Twisted
```

We need `coverage` only for the unit tests, while `Twisted` is needed for both the unit and functional tests. The `pylint` strict dependency ensures that as `pylint` adds more rules, our code does not acquire new test failures. This does mean we need to update `pylint` manually from time to time.

```
commands =
    {py36,py27,pypy}-unit: python -Wall \
                                  -Wignore::DeprecationWarning \
                                  -m coverage \
                                  run -m twisted.trial \
                                  --temp-directory build/_trial_temp \
                                  {posargs:ncolony}
```

```
    {py36,py27,pypy}-unit: coverage report --include ncolony* \
                           --omit */tests/*,*/interfaces*,*/_version* \
↪                      --show-missing --fail-under=100
    py27-lint: pylint --rcfile admin/pylintrc ncolony
    py27-lint: python -m ncolony tests.nitpicker
    py27-lint: flake8 ncolony
    {py36,py27,pypy}-func: python -Werror -W ignore::DeprecationWarning \
                                  -W ignore::ImportWarning \
                                  -m ncolony tests.functional_test
```

Configuring "one big test environment" means we need to have all our commands mixed in one bag and select based on patterns. This is also a more realistic test run command – we want to run with warnings enabled, but disable warnings we do not worry about, and also enable code coverage testing. While the exact complications will vary, we almost always need enough things so that the commands will grow to a decent size.

```
[testenv:py27-wheel]
skip_install = True
deps =
      coverage
      Twisted
      wheel
      gather
commands =
      mkdir -p {envtmpdir}/dist
      pip wheel . --no-deps --wheel-dir {envtmpdir}/dist
      sh -c "pip install --no-index {envtmpdir}/dist/*.whl"
      coverage run {envbindir}/trial \
               --temp-directory build/_trial_temp {posargs:ncolony}
      coverage report --include */site-packages/ncolony* \
                      --omit */tests/*,*/interfaces*,*/_version* \
                      --show-missing --fail-under=100
```

The py27-wheel test run ensures we can build, and test, a wheel. As a side effect, this means a complete test run will build a wheel. This allows us to upload a *tested* wheel to PyPI when it is release time.

```
[testenv:docs]
changedir = docs
deps =
    sphinx
    Twisted
commands =
    sphinx-build -W -b html -d {envtmpdir}/doctrees . {envtmpdir}/html
basepython = python2.7
```

The documentation build is one of the reasons why Tox shines. It only installs `sphinx` in the virtual environment for building documentation. This means that an undeclared dependency on `sphinx` would make the unit tests fail, since `sphinx` is not installed there.

2.5 Pipenv and Poetry

Pipenv and Poetry are two new ways to produce Python projects. They are inspired by tools like `yarn` and `bundler` for JavaScript and Ruby, respectively, which aim to encode a more complete development flow. By themselves, they are not a replacement for Tox – they do not encode the ability to run with multiple Python interpreters, or completely override the dependency. However, it is possible to use them, in tandem with a CI-system configuration file, like `Jenkinsfile` or `.circleci/config.yml`, to build against multiple environments.

However, their main strength is in allowing easier interactive development. This is useful, sometimes, for more exploratory programming.

2.5.1 Poetry

The easiest way to install poetry is to use `pip install --user poetry`. However, this will install all of its dependencies into your user environment, which has the potential to make a mess of things. One way to do it in a clean way is to create a dedicated virtual environment.

```
$ python3 -m venv ~/.venvs/poetry
$ ~/.venvs/poetry/bin/pip install poetry
$ alias poetry=~/.venvs/poetry/bin/poetry
```

This is an example of using an *unactivated* virtual environment.

The best way to use poetry is to create a dedicated virtual environment for the project. We will build a small demo project. We will call it "useful."

```
$ mkdir useful
$ cd useful
$ python3 -m venv build/useful
$ source build/useful/bin/activate
(useful)$ poetry init
(useful)$ poetry add termcolor
(useful)$ mkdir useful
(useful)$ touch useful/__init__.py
(useful)$ cat > useful/__main__.py
import termcolor
print(termcolor.colored("Hello", "red"))
```

If we have done all this, running *python -m useful* in the virtual environment will print a red `Hello`. After we have interactively tried various colors, and maybe decided to make the text bold, we are ready to release:

```
(useful)$ poetry build
(useful)$ ls dist/
useful-0.1.0-py2.py3-none-any.whl useful-0.1.0.tar.gz
```

2.5.2 Pipenv

Pipenv is a tool to create virtual environments that match a specification, in addition to ways to evolve the specification. It relies on two files: `Pipfile` and `Pipfile.lock`. We can install `pipenv` similarly to how we installed `poetry` – in a custom virtual environment and add an alias.

In order to start using it, we want to make sure no virtual environments are activated. Then,

```
$ mkdir useful
$ cd useful
$ pipenv add termcolor
```

```
$ mkdir useful
$ touch useful/__init__.py
$ cat > useful/__main__.py
import termcolor
print(termcolor.colored("Hello", "red"))
$ pipenv shell
(useful-hwA3o_b5)$ python -m useful
```

This will leave in its wake a `Pipfile` that looks like this:

```
[[source]]
url = "https://pypi.org/simple"
verify_ssl = true
name = "pypi"

[packages]
termcolor = "*"

[dev-packages]

[requires]
python_version = "3.6"
```

Note that in order to package `useful`, we still have to write a `setup.py`. Pipenv limits itself to managing virtual environments, and it does consider building and publishing separate tasks.

2.6 DevPI

DevPI is a PyPI-compatible server that can be run locally. Though it does not scale to PyPI-like levels, it can be a powerful tool in a number of situations.

DevPI is made up of three parts. The most important one is `devpi-server`. For many use cases, this is the only part that needs to run. The server serves, first and foremost, as a caching proxy to PyPI. It takes advantage of the fact that packages on PyPI are *immutable*: once we have a package, it can never change.

There is also a web server that allows us to search in the local package directory. Since a lot of use cases do not even involve searching on the PyPI website, this is definitely optional. Finally, there is a client command-line tool that allows configuring various parameters on the running instance. The client is most useful in more esoteric use cases.

Installing and running DevPI is straightforward. In a virtual environment, simply run:

```
(devpi)$ pip install devpi-server
(devpi)$ devpi-server --start --init
```

The `pip` tool, by default, goes to `pypi.org`. For some basic testing of DevPI, we can create a new virtual environment, `playground`, and run:

```
(playground)$ pip install \
              -i http://localhost:3141/root/pypi/+simple/ \
              httpie glom
(playground)$ http --body https://httpbin.org/get | glom '{"url":"url"}'
{
  "url": "https://httpbin.org/get"
}
```

Having to specify the `-i ...` argument to `pip` every time would be annoying. After checking that everything worked correctly, we can put the configuration in an environment variable:

```
$ export PIP_INDEX_URL=http://localhost:3141/root/pypi/+simple/
```

Or to make things more permanent:

```
$ mkdir -p ~/.pip && cat > ~/.pip/pip.conf << EOF
[global]
index-url = http://localhost:3141/root/pypi/+simple/

[search]
index = http://localhost:3141/root/pypi/
```

The above file location works for UNIX operating systems. On Mac OS X the configuration file is `$HOME/Library/Application Support/pip/pip.conf`. On Windows the configuration file is `%APPDATA%\pip\pip.ini`.

DevPI is useful for disconnected operations. If we need to install packages without a network, DevPI can be used to cache them. As mentioned earlier, virtual environments are disposable and often treated as mostly immutable. This means that a virtual environment with the right packages *is not* a useful thing without a network. The chances are high that some situation or the other will either require or suggest creating it from scratch.

However, a caching server is a different matter. If all package retrieval is done through a caching proxy, then destroying a virtual environment and rebuilding it is fine, since the source of truth is the package cache. This is as useful for taking a laptop into the woods for disconnected development as it is for maintaining proper firewall boundaries and having a consistent record of all installed software.

In order to "warm up" the DevPI cache, that is, make sure it contains all needed packages, we need to use `pip` to install them. One way to do it is, after configuring DevPI and `pip`, is to run `tox` against a source repository of software under development. Since `tox` goes through all test environments, it downloads all needed packages.

It is definitely a good practice to also preinstall in a disposable virtual environment any `requirements.txt` that are relevant.

However, the utility of DevPI is not limited to disconnected operations. Configuring one inside your build cluster, and pointing the build cluster at it, completely avoids the risk for a "leftpad incident," where a package you rely on gets removed by the author from PyPI. It might also make builds faster, and it will definitely cut out a lot of outgoing traffic.

Another use for DevPI is to test uploads, before uploading them to PyPI. Assuming `devpi-server` is already running on the default port, we can:

```
(devpi)$ pip install devpi-client twine
(devpi)$ devpi use http://localhost:3141
(devpi)$ devpi user -c testuser password=123
(devpi)$ devpi login testuser --password=123
(devpi)$ devpi index -c dev bases=root/pypi
(devpi)$ devpi use testuser/dev
(devpi)$ twine upload --repository http://localhost:3141/testuser/dev \
             -u testuser -p 123 my-package-18.6.0.tar.gz
(devpi)$ pip install -i http://localhost:3141/testuser/dev my-package
```

Note that this allows us to upload to an index that we only use explicitly, so we are not shadowing `my-package` for all environments that are not using this explicitly.

An even more advanced use-case, we can do this:

```
(devpi)$ devpi index root/pypi mirror_url=https://ourdevpi.local
```

This will make our DevPI server a mirror of a local, "upstream," DevPI server. This allows us to upload private packages to the "central" DevPI server, in order to share with our team. In those cases, the upstream DevPI server will often need to be run behind a proxy – and we need to have some tools to properly manage user access.

Running a "centralized" DevPI behind a simple proxy that asks for username and password allows an effective private repository. For that, we would first want to remove the `root/pypi` index:

```
$ devpi index --delete root/pypi
```

and then re-create it with

```
$ devpi index --create root/pypi
```

This means the root index no longer will mirror `pypi`. We can upload packages now directly to it. This type of server is often used with the argument `--extra-index-url` to `pip`, to allow `pip` to retrieve both from the private repository and the external one. However, sometimes it is useful to have a `DevPI` instance that only serves specific packages. This allows enforcing rules about auditing before using any packages. Whenever a new package is needed, it is downloaded, audited, and then added to the private repository.

2.7 Pex and Shiv

While it is currently nontrivial to compile a Python program into one self-contained executable, we can do something that is *almost* as good. We can compile a Python program into a single file that only needs an installed interpreter to run. This takes advantage of the particular way Python handles startup.

When running `python /path/to/filename`, Python does two things:

- Adds the directory `/path/to` to the module path.
- Executes the code in `/path/to/filename`.

When running python/path/to/directory/, Python will behave exactly as though we typed python/path/to/directory/__main__.py.

In other words, Python will do the following two things:

- Add the directory /path/to/directory/ to the module path.
- Executes the code in /path/to/directory/__main__.py.

When running python /path/to/filename.zip, Python will treat the file as a directory.

In other words, Python will do the following two things:

- Add the "directory" /path/to/filename.zip to the module path.
- Executes the code in the __main__.py it extracts from /path/to/filename.zip.

Zip is an end-oriented format: The metadata, and pointers to the data, are all at the end. This means that adding a prefix to a zip file does not change its contents.

So, if we take a zip file, and prefix it with #!/usr/bin/python<newline>, and mark it executable, then when running it, Python will be running a zip file. If we put the right bootstrapping code in __main__.py, and put the right modules in the zip file, we can get all of our third-party dependencies in one big file.

Pex and Shiv are tools for producing such files, but they both rely on the same underlying behavior of Python and of zip files.

2.7.1 Pex

Pex can be used either as a command-line tool or as a library. When using it as a command-line tool, it is a good idea to prevent it from trying to do dependency resolution against PyPI. All dependency resolution algorithms are flawed in some way. However, due to pip's popularity, packages will explicitly work around flaws in its algorithm. Pex is less popular, and there is no guarantee that packages will try explicitly to work with it.

The safest thing to do is to use pip wheel to build all wheels in a directory and then tell Pex to use only this directory.

For example,

```
$ pip wheel --wheel-dir my-wheels -r requirements.txt
$ pex -o my-file.pex --find-links my-wheels --no-index \
      -m some_package
```

Pex has a few ways to find the entry point. The two most popular ones are -m some_package, which will behave as though python -m some_package; or -c console-script, which will find what script would have been installed as console-script, and invoke the relevant entry point.

It is also possible to use Pex as a library.

```
from pex import pex_builder
```

Most of the logic to build Pex files is in the pex_builder module.

```
builder = pex_builder.PEXBuilder()
```

We create a builder object.

```
builder.set_entry_point('some_package')
```

We set the entry point. This is equivalent to the -m some_package argument on the command line.

```
builder.set_shebang(sys.executable)
```

The Pex binary has a sophisticated argument to determine the right shebang line. This is sometimes specific to the expected deployment environment, so it is a good idea to put some thought into the right shebang line. One option is /usr/bin/env python, which will find what the current shell calls python. It is sometimes a good idea to specify a version here, such as /usr/local/bin/python3.6, for example.

```
subprocess.check_call([sys.executable, '-m', 'pip', 'wheel',
                       '--wheel-dir', 'my-wheels',
                       '--requirements', 'requirements.txt'])
```

Once again, we create wheels with pip. As tempting as it is, pip is not usable as a library, so shelling out is the only supported interface.

```
for dist in os.listdir('my-wheels'):
    dist = os.path.join('my-wheels', dist)
    builder.add_dist_location(dist)
```

We add all packages that pip built.

```
builder.build('my-file.pex')
```

Finally, we have the builder produce a Pex file.

2.7.2 Shiv

Shiv is a modern take on the same ideas behind Pex. However, since it uses `pip` directly, it needs to do a lot less itself.

```
$ shiv -o my-file.shiv -e some_package -r requirements.txt
```

Because shiv just offloads to pip actual dependency resolution, it is safe to call it directly. Shiv is a younger alternative to Pex. This means a lot of cruft has been removed, but it is still lacking somewhat in maturity.

For example, the documentation for command-line arguments is a bit thin. There is also no way to currently use it as a library.

2.8 XAR

XAR (eXecutable ARchive) is a generic format for shipping self-contained executables. While not being Python specific, it is designed as Python first. It is natively installable via PyPI, for example.

The downsides of XAR is that it assumes a certain level of system support for `fuse` (Filesystem in User SpacE) that is not universal yet. This is not a problem if all machines designed to run the XAR, Linux, or Mac OS X are under your control. The instructions for how to install proper FUSE support are not complex, but they do require administrative privileges. Note that XAR is also less mature than Pex.

However, assuming proper SquashFS support, many other concerns vanish: including, most importantly, compared to `pex` or `shiv`, local Python versions. This makes XAR an interesting choice for either shipping developer tools or local system management scripts.

In order to build a XAR, we can call `setup.py` with `bdist_xar`, if `xar` is installed.

```
python setup.py bdist_xar --console-scripts=my-script
```

In this example, `my-script` is the name of a console script entry point, specified in the `setup.py` with the following:

```
entry_points=dict(
   console_scripts=["my-script = package.module:function"],
)
```

In some cases, the `--console-scripts` argument is not necessary. If, as in the example above, there is only one console script entry point, then it is implied. Otherwise, if there is a console script with the same name as the package, then that one is used. This accounts for quite a few cases, which means this argument is often redundant.

2.9 Summary

Much of the power of Python comes from its powerful third-party ecosystems: whether for data science or networking code, there are many good options. Understanding how to install, use, and update third-party packages are crucial to using Python well.

With private package repositories, using Python packages for internal libraries, and distributing them in a way compatible with open source libraries, is often a good idea. It allows using the same machinery for internal distribution, versioning, and dependency management.

CHAPTER 3

Interactive Usage

Python is often used for exploratory programming. Often, the final result is not the program but an answer to a question. For scientists, the question might be "what is the likelihood of a medical intervention working?" For people troubleshooting computers, the question might be "which log file has the message I need?"

However, regardless of the question, Python can often be a powerful tool to answer it. More importantly, in exploratory programming, we expect to encounter more questions, based on the answer.

The interactive model in Python comes from the original Lisp environments' "Read-Eval-Print Loop" (REPL, for short). The environment reads a Python expression, evaluates it in an environment that persists in memory, prints the result, and loops back.

The REPL environment native to Python is popular because it is built in. However, a few third-party REPL tools are even more powerful, built to do things that the native one could not or would not. These tools give a powerful way to interact with the operating system, exploring and molding until the desired state is achieved.

3.1 Native Console

Launching `python` without any arguments will open up the "interactive console." It is a good idea to use `pyenv` or a virtual environment to make sure that the Python version is up to date.

The availability of an interactive console immediately, without installing anything else, is one reason why Python is suited for exploratory programming. We can immediately ask questions.

The questions can be trivial:

```
>>> 2 + 2
4
```


M. Zadka, *DevOps in Python*, https://doi.org/10.1007/978-1-4842-4433-3_3

They can be used to calculate sales tax in the Bay area:

```
>>> rate = 9.25
>>> price = 5.99
>>> after_tax = price * (1 + rate / 100.)
>>> after_tax
6.544075
```

Or they can answer important questions about the operating environment:

```
>>> import os
>>> os.path.isfile(os.path.expanduser("~/.bashrc"))
True
```

Using the Python native console without readline is unpleasant. Rebuilding Python with readline support is a good idea, so that the native console will be useful. If this is not an option, using one of the alternative consoles is recommended. For example, a locally built Python with specific options might not include readline, and it might be problematic to redistribute a new Python to the entire team.

If readline support is installed, Python will use it to support line editing and history. It is also possible to save the history using readline.write_history_file. This is often after having used the console for a while, in order to have a reference for what has been done, or to copy whatever ideas worked into a more permanent form.

When using the console, the _ variable will have the value of the last expression-statement evaluated. Note that exceptions, statements that are not expressions and statements stat are expressions that evaluate to None will not change the value of _. This is useful during an interactive session when only after having seen the representation of the value, do we realize we needed that as an object.

```
>>> import requests
>>> requests.get("http://en.wikipedia.org")
<Response [200]>
>>> a=_
>>> a.text[:50]
'<!DOCTYPE html>\n<html class="client-nojs" lang="en'
```

Only after using the `.get` function, we realize that what we actually wanted was the text. Luckily, the `Response` object is saved in the variable `_`. We put the value of the variable in a immediately, _ is replaced quickly. As soon as we evaluate `a.text[:50]`, _ is now a 50-character string. If we had not saved _ in a variable, all but the first 50 characters would have been lost.

Notice that this _ convention is kept by every good Python REPL, and so the trick to “keep returned values in one-letter variables” is often useful when doing explorations.

3.2 The Code Module

The `code` module allows us to run our own interactive loop. An example of when this can be useful is when running commands with a special flag, we can drop into a prompt at a specific point. This allows us to have a REPL environment after we have set things up in a certain way. This holds for both inside the interpreter, setting up the namespace with useful things; and in the external environment, perhaps initializing files or setting up external services.

The highest-level use of `code` is the `interact` function.

```
>>> import code
>>> code.interact(banner="Welcome to the special interpreter",
...               local=dict(special=[1, 2, 3]))
Welcome to the special interpreter
>>> special
[1, 2, 3]
>>> ^D
now exiting InteractiveConsole...
>>> special
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'special' is not defined
```

This shows an example of running a REPL loop with a variable `special`, which sets to a short list.

For the lowest-level use of `code`, if you want to own the UI yourself, `code.compile_command(source, [filename="<input>"], symbol="single")` will return a code object (that can be passed to `exec`), `None` if the command is incomplete or raise `SyntaxError`, `OverflowError`, or `ValueError` if there is a problem with the command.

The `symbol` argument should almost always be `"single."` The exception is if the user is prompted to enter code that will evaluate to an expression (for example, if the value is to be used by the underlying system). In that case, `symbol` should be set to `"eval."`

This allows us to manage interacting with the user ourselves. It can be integrated with a UI, or a remote network interface, to allow interactivity in any environment.

3.3 ptpython

The `ptpython` tool, short for "prompt toolkit Python," is an alternative to the built-in REPL. It uses the prompt toolkit for console interaction, instead of `readline`.

Its main advantage is the simplicity of installation. A simple `pip install ptpython` in a virtual environment and regardless of readline build problems, a high-quality Python REPL appears.

`ptpython` supports completion suggestions, multiline editing, and syntax highlighting.

On startup, it will read `~/.ptpython/config.py`. This means it is possible to locally customize ptpython in arbitrary ways. The way to configure is to implement a function, `configure`, which accepts an object (of type `PythonRepl`) and mutates it.

There are a lot of possibilities, and sadly the only real documentation is the source code. The relevant reference `__init__` is `ptpython.python_input.PythonInput`. Note that `config.py` really is an arbitrary Python file. Therefore, if you want to distribute modifications internally, it is possible to distribute a local PyPI package and have people import a `configure` function from it.

3.4 IPython

IPython, which is also the foundation of Jupyter, which will be covered later, is an interactive environment whose roots are in the scientific computing community. IPython is an interactive command prompt, similar to the `ptpython` utility or Python's native REPL.

However, it aims to give a sophisticated environment. One of the things that it does is number every input and output from the interpreter. It is useful to be able to refer to those numbers later. IPython puts all inputs in the `In` array and outputs in the `Out` array. This allows for nice symmetry: if IPython says `In[4]`, for example, this is how to access that value.

```
In [1]: print("hello")
hello

In [2]: In[1]
Out[2]: 'print("hello")'

In [3]: 5 + 4.0
Out[3]: 9.0

In [4]: Out[3] == 9.0
Out[4]: True
```

It also supports tab completion out of the box. IPython uses both its own completion and the `jedi` library for static completion.

It also supports built-in help. Typing `var_name?` will try to find the best context-relevant help for the object in the variable, and display it. This works for functions, classes, built-in objects, and more.

```
In [1]: list?
Init signature: list(self, /, *args, **kwargs)
Docstring:
list() -> new empty list
list(iterable) -> new list initialized from iterable's items
Type:           type
```

IPython also supports something called "magic," where prefixing a line with `%` will execute a magic function. For example, `%run` will run a Python script inside the current namespace. As another example, `%edit` will launch an editor. This is useful if during usage, a statement needs more sophisticated editing.

In addition, prefixing a line with `!` will run a system command. One useful way to take advantage of this is `!pip install something`. This is why it is useful to install IPython inside virtual environments that are used for interactive development.

IPython can be customized in a number of ways. While in an interactive session, the `%config` magic command can be used to change any option. For example, `%config InteractiveShell.autocall = True` will set the autocall option, which means expressions that are callable are called, even without parentheses. This is moot for any options that only affect startup. We can change these options, as well as any others, using the command line. For example, `ipython --InteractiveShell.autocall=True`, will launch into an autocalling interpreter.

If we want custom logic to decide on configuration, we can run IPython from a specialized Python script.

```
from traitlets import config
import IPython

my_config = config.Config()
my_config.InteractiveShell.autocall = True

IPython.start_ipython(config=my_config)
```

If we package this in a dedicated Python package, we can distribute it to a team using either PyPI or a private package repository. This allows having a homogenous custom IPython configuration for a development team.

Finally, configuration can also be encoded in profiles, which are Python snippets located under `~/.ipython` by default. The profile directory can be modified by an explicit command-line parameter `--ipython-dir` or an environment variable `IPYTHONDIR`.

3.5 Jupyter Lab

Jupyter is a project that uses web-based interaction to allow for sophisticated exploratory programming. It is not limited to Python, though it does originate in Python. The name stands for “Julia/Python/R,” the three languages most popular for exploratory programming, especially in data science.

Jupyter Lab, the latest evolution of Jupyter, was originally based on IPython. It now sports a full-featured web interface and a way to remotely edit files. The main users of Jupyter tend to be scientists. They take advantage of the ability to see how results were derived to add in reproducibility and peer review.

Reproducibility and peer review are also important for DevOps work. The ability to show the steps that led to deciding which list of hosts to restart, so that it can be regenerated if circumstances change, for example, is highly useful. The ability to attach a notebook, detailing the steps that were taken during an outage, together with the output from the steps, to a postmortem analysis can aid in understanding what happened, and how to avoid a problem in the future or recover from it more effectively.

It is important to note here that notebooks are *not* an auditability tool: they can be executed out of order and have blocks modified and re-executed. However, properly used, they allow us to record what has been done.

Jupyter allows true exploratory programming. This is useful for scientists, who might not understand the true scope of a problem beforehand.

It is important to note here that notebooks are *not* an auditability tool: they can be executed out of order and have blocks modified and re-executed. However, properly used, they allow us to record what has been done. This is also useful for systems integrators, faced with complex systems, where it is hard to predict where the problem lies before exploration, either.

Installing Jupyter Lab in a virtual environment is a simple matter of doing `pip install jupyterlab`. When starting `jupyter lab`, by default, it will start a web server on an open port starting at `8888` and attempt to launch a web browser to watch it. If working on an environment that is "too interesting" (for example, the default web browser is not configured properly), the standard output will contain a preauthorized URL to access the server. If all else fails, it is possible to copy-paste the token printed to standard output into the browser after manually entering the URL in a web browser. It is also possible to access the token with `jupyter notebook` list, which will list all currently running servers.

Once inside Jupyter Lab, there are five things we can launch:

- Console
- Terminal
- Text editor
- Notebook
- Spreadsheet editor

The Console is a web-based interface to IPython. All that was said about IPython previously (for example, the In and Out arrays). The Terminal is a full-fledged terminal emulator in the browser. This is useful for a remote terminal inside a VPN: all it needs as far as connectivity needs is an open web port, and it can also be protected in the regular ways that web ports are protected: TLS, client-side certificates, and more. The text editor is useful for editing remote files. This is an alternative to running a remote shell, and an editor such as vi in it. It has the advantage of avoiding UI lag, while still having full file-editing capabilities.

The most interesting thing to launch, though, is a notebook: indeed, many a session will using nothing but notebooks. A notebook is a JSON file that records a session. As the session unfolds, Jupyter will save "snapshots" of the notebook, as well as the latest version. A notebook is made of a sequence of Cells. The two most popular cell types are "code" and "Markdown." A "code" cell type will contain a Python code snippet. It will execute it in the context of the session's namespace. The namespace is persistent from one cell execution to the other, corresponding to a "kernel" running. The kernel accepts cell content using a custom protocol, interprets them as Python, executes them, and returns both whatever was returned by the snippet as well as output from this.

When launching a Jupyter server, by default, it will use the local IPython kernel as its only possible kernel. This means that the server will, for example, only be able to use the same Python version and the same set of packages. However, it is possible to connect a kernel from a different environment to this server. The only requirement is that the environment has the ipykernel package installed. From the environment, run:

```
python -m ipykernel install \
       --name my-special-env \
       --display-name "My Env"
       --prefix=$DIRECTORY
```

Then, from the Jupyter server environment, run:

```
jupyter kernelspec install $DIRECTORY/jupyter/kernels/my-special-env
```

This will cause the Jupyter server in this environment to support the kernel from the special environment. This allows running one semipermanent Jupyter server and connecting kernels from any environment that is "interesting": installing specific modules, running a specific version of Python, or any other difference. One other usage of alternative kernels, which will not be covered here in details, is alternative languages.

Julia and R kernels are supported upstream, but there exist third-party kernels for many languages – even `bash`!

Jupyter supports all magic commands from IPython. Especially useful, again, is the `!pip install ...` command to install new packages in the virtual environment. Especially if being careful, and installing precise dependencies, this makes a notebook be high-quality documentation of how to achieve a result in a way that is replayable.

Since Jupyter is one level of indirection away from the kernel, we can restart the kernel directly from Jupyter. This means the whole Python process gets restarted, and any in-memory results are gone. We can re-execute cells in any order, but there is a single-button way to execute all cells in order. Restarting the kernel, and executing all cells in order, is a nice way of "testing" a notebook for working conditions – although, naturally, any effects on the external world will not be reset.

Jupyter notebooks are useful as attachments to tickets and postmortems, both as a way of documenting specific remediations, as well as documenting "state of things" by running query APIs and collecting the results in the notebook. Usually, when attaching a notebook in such a way, it is useful to also export it to a more easily readable format, such as HTML or PDF, and attach that as well. However, more and more tools integrate direct notebook viewing, making this step redundant. For example, GitHub projects and Gists already render notebooks directly.

Alongside the notebooks, Jupyter Lab sports a rudimentary, but functional, browser-based remote development environment. The first part is a remote file manager. Among other things, this allows uploading and downloading files. One use for it, among many, is the ability to upload notebooks from the local computer and download them back again. There are better ways to manage notebooks, but in a pinch, being able to retrieve a notebook is extremely useful. Similarly, any persistent outputs from Jupyter, such as processed data files, images, or charts, can also be downloaded.

Next, alongside the notebooks, is a remote IPython console. Though of limited use next to the notebook, there are still some cases where using the console is easier. A session that requires a lot of short commands can be more keyboard-centric by using the IPython console, and thus more efficient.

There is also a file editor. Although it is a far cry from being a full-fledged developer editor, lacking thorough code understanding and completion, it is often useful in a pinch. It allows directly editing files on the remote Jupyter host. One use case is directly fixing library code that the notebook is using, and then restarting the kernel. While

integrating it into a development flow takes some care, as an emergency measure to fix and continue, this is invaluable.

Last, there is a remote browser-based terminal. Between the terminal, the file editor and the file manager, a running Jupyter server allows complete browser-based remote access and management, even before thinking of the notebooks. This is important to remember for security implications, but it is also a powerful tool whose various uses we will explore later. For now, suffice it to say, the power that using a Jupyter notebook brings to remote system administration tasks is hard to overestimate.

3.6 Summary

The faster the feedback cycle, the faster we can deploy new, tested, solutions. Using Python interactively allows getting the quickest possible feedback: immediate.

This is often useful to clarify a library's documentation, a hypothesis about a running system, or just your understanding of Python.

The interactive console is also a powerful control panel from which to launch computations when the end result is not well understood: for example, when debugging the state of software systems.

CHAPTER 4

OS Automation

Python was initially built to automate a distributed operating system called an "amoeba." Though the Amoeba OS is mostly forgotten, Python has found a home in automating UNIX-like operating systems tasks.

Python wraps the traditional UNIX C API lightly, giving full access to the system calls that run UNIX while making them just a little safer to use: an approach that was dubbed "C with foam padding." This willingness to wrap low-level operating system APIs has made it a good choice for the wide berth between the programs that UNIX shell is good for, and the programs the C programming language is good for.

As the saying goes, with great power comes great responsibility. In order to allow for programmer power and flexibility, Python does not stop programmers from wreaking havoc. Carefully using Python to write programs that work and, more importantly, break in predictable, safe ways is a skill that is worth mastering.

4.1 Files

It has been a long time since "everything is a file" was an accurate mantra on UNIX. Nevertheless, many things are files, and even more things are enough like files that manipulating them with file-based system calls works.

When dealing with files' contents, Python programs can go down one of two routes. They can open them as "text" or as "binary." Although files themselves are neither text nor binary, just a blob of bytes, the opening mode is important.

When opening a file as binary, the bytes are read and written as is – as byte strings. This is useful with files that are non-textual, such as picture files.

When opening a file as text, an *encoding* has to be used. It can be specified explicitly, but in certain situations, defaults apply. All bytes read from the file are decoded, and the code receives a *character* string. All strings written to the file are encoded to bytes. This means the interface with the file is with strings – sequences of characters.

M. Zadka, *DevOps in Python*, https://doi.org/10.1007/978-1-4842-4433-3_4

A simple example of a binary file is the GIMP "XCF" internal format. GIMP is an image manipulation program, and it saves files in its internal XCF format with more details than images have. For example, layers in the XCF will be separate, for easy editing.

```
>>> with open("Untitled.xcf", "rb") as fp:
...     header = fp.read(100)
```

Here we open a file. The `rb` argument stands for "read, binary." We read the first hundred bytes. We will need far fewer, but this is often a useful tactic. Many files have some metadata at the beginning.

```
>>> header[:9].decode('ascii')
'gimp xcf '
```

The first nine characters can actually be decoded to ASCII text, and happen to be the name of the format.

```
>>> header[9:9+4].decode('ascii')
'v011'
```

The next four characters are the version. This file is the 11th version of XCF.

```
>>> header[9+4]
0
```

A 0 byte finishes the "what is this file" metadata. This has various advantages.

```
>>> struct.unpack('>I', header[9+4+1:9+4+1+4])
(1920,)
```

The next four bytes are the width, as a number in big-endian format. The `struct` module knows how to parse these. The `>` says it is big endian, and the `I` says it is an unsigned 4-byte integer.

```
>>> struct.unpack('>I', header[9+4+1+4:9+4+1+4+4])
(1080,)
```

The next four bytes are the width. This simple code gave us the high-level data: it confirmed that this is XCF, it showed what version of the format it is, and we could see the dimensions of the image.

When opening files as text, the default encoding is UTF-8. One advantage of UTF-8 is that it is designed to fail quickly if something is *not* UTF-8: it is carefully designed to fail on ISO-8859-[1-9], which predates Unicode, as well as on most binary files. It is also backwards compatible with ASCII, which means pure ASCII files will still be valid UTF-8.

The most popular way to parse text files is line by line, and Python supports that by having an open text file be an iterator that yields the lines in order.

```
>>> fp = open("things.txt", "w")
>>> fp.write("""\
... one line
... two lines
... red line
... blue line
... """)
39
>>> fp.close()
>>> fpin = open("things.txt")
>>> next(fpin)
'one line\n'
>>> next(fpin)
'two lines\n'
>>> next(fpin)
'red line\n'
>>> next(fpin)
'blue line\n'
>>> next(fpin)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Usually we will not call `next` directly, but use `for`. Additionally, usually we use files as context managers, to make sure they close at a well-understood point. However, especially in REPL scenarios, there is a trade-off: opening the file without a context manager allows us to explore reading bits and pieces.

Files on a unix system are more than just blobs of data. They have various metadata attached, which can be queried and sometimes changed.

The rename system call is wrapped in the os.rename Python function. Since rename is atomic, this can help implement operations that require a certain state.

In general, note that the os module tends to be a thin wrapper over operating system calls. The discussion here is relevant to UNIX-like systems: Linux, BSD-based systems, and, for the most part, Mac OS X. It is worth keeping in mind, but it is not worth pointing each place where we are making UNIX-specific assumptions.

For example,

```
with open("important.tmp", "w") as fout:
    fout.write("The horse raced past the barn")
    fout.write("fell.\n")
os.rename("important.tmp", "important")
```

This ensures that when reading the important file, we do not accidentally misunderstand the sentence. If the code crashes in the middle, instead of believing that the horse raced past the barn, we get nothing from important. We only rename important.tmp to important at the end, after the last word has been written to the file.

The most important example of a file-which-is-not-a-blob, in UNIX, is a directory. The os.makedirs function allows us to ensure a directory exists easily with

```
os.makedirs(some_path, exists_ok=True)
```

This combines powerfully with the path operations from os.path to allow safe creation of a nested file:

```
def open_for_write(fname, mode=""):
    os.makedirs(os.path.dirname(fname), exists_ok=True)
    return open(fname, "w" + mode)

with open_for_write("some/deep/nested/name/of/file.txt") as fp:
    fp.write("hello world")
```

This can come in useful, for example, when mirroring an existing file layout.

The os.path module has, mostly, string manipulation functions that assume strings are file names. The dirname function returns the directory name, so os.path.dirname("a/b/c") would return a/b. Similarly, the function basename returns the "file name," so os.path.basename("a/b/c") would return c. The inverse of both is the os.path.join function, which join paths: os.path.join("some", "long/and/winding", "path") would return some/long/and/finding/path.

Another set of functions in the `os.path` module has a slightly higher-level abstraction for getting file metadata. It is important to note that these functions are often light wrappers around operating system functionality, and *do not* try to hide operating system quirks. This means that operating system quirks can "leak" through the abstraction.

The biggest metadata is `os.path.exists`: does the file exist? This comes in handy sometimes, though often it is better to write code in a way that is agnostic of file existence: file existence can have races. Subtler are the `os.path.is...` functions: `isdir`, `isfile`, `islink`, and more can decide if a file name points to what we expect.

The `os.path.get...` functions get non-boolean metadata: access time, modification time, c-time (sometimes shortened to "creation time," but misleadingly not the actual time of creation in a set of subtle circumstances, and more accurately refer to as "i-node modification time"), and `getsize` getting the size of the file.

The `shutil` module ("shell utilities") contains some higher-level operations. `shutil.copy` will copy a file's contents as well as metadata. `shutil.copyfile` will copy contents only. `shutil.rmtree` is the equivalent of `rm -r`, while `shutil.copytree` is the equivalent of `cp -r`.

Finally, temporary files are often useful. Python's `tempfile` module produces temporary files that are secure and resistant to leaks. The most useful functionality is `NamedTemporaryFile`, which can be used as a context.

A typical usage looks like this:

```
with NamedTemporaryFile() as fp:
    fp.write("line 1\n")
    fp.write("line 2\n")
    fp.flush()
    function_taking_file_name(fp.name)
```

Note that the `fp.flush` there is important. The file object caches write until closed. However, `NamedTemporaryFile` will vanish when closed. Explicitly flushing it is important before calling a function that will reopen the file for reading.

4.2 Processes

The main module to deal with running subprocesses in Python is `subprocess`. It contains a high-level abstraction that matches the intuitive model most have when they think of "running commands," rather than the low-level model implemented in UNIX, using `exec` and `fork`.

It is also a powerful alternative to calling the `os.system` function, which is problematic in several ways. For one, `os.system` spawns an *extra* process, the shell. This means that it depends on the shell, which on some weirder installation can differ with a more "exotic" system shell like `ash` or `fish`. Finally, it means that the shell will *parse* the string, which means the string has to be properly serialized. This is a hard task to do, since the formal specification for the shell parser is long. Unfortunately, it is *not* hard to write something that will work fine most of the time, so most bugs are subtle and break at the worst possible time. This sometimes even manifests as a security flaw.

While `subprocess` is not *completely* flexible, for most needs, this module is perfectly adequate.

`subprocess` itself is also divided into high-level functions and a lower-level implementation level. The high-level functions, which should be used in most circumstances, are `check_call` and `check_output`. Among other benefits, they behave like running a shell with `-e`, or `set err` – they will immediately raise an exception if a command returns with a non-zero value.

The slightly lower-level is `Popen`, which creates processes and allows fine-grained configuration of their inputs and outputs. Both `check_call` and `check_output` are implemented on top of `Popen`. Because of that, they share some semantics and arguments. The most important argument is `shell=True`, and it is most important in that it is almost always a bad idea to use it. When the argument is given, a string is expected, and is passed to the shell to parse it.

Shell parsing rules are subtle and full of corner cases. If it is a constant command, there is no benefit there: We can translate the command to separate arguments in code. If it includes some input, it is almost impossible to reliably escape it in a way that makes it impossible to introduce an injection problem. On the other hand, without this, creating commands on the fly is reliable, even in the face of potentially hostile inputs.

The following, for example, will add a user to the docker group.

```
subprocess.check_call(["usermod", "-G", "docker", "some-user"])
```

Using `check_call` means that if the command fails for some reason, such as the user not existing, this will automatically raise an exception. This avoids a common failure mode, where scripts do not report accurate status.

If we want to make it into a function that takes a username, it is straightforward:

```
def add_to_docker(username):
    subprocess.check_call(["usermod", "-G", "docker", username])
```

Note that this is safe to call even if the argument contains spaces, #, or other characters with special meanings.

In order to tell which groups the current user is currently in, we can run groups.

```
groups = subprocess.check_output(["groups"]).split()
```

Again, this will automatically raise an exception if the command fails. If it succeeds, we get the output as a string: no need to manually read and determine end conditions.

Both of these functions get common arguments. cwd allows running a command inside of a given directory. This matters for commands that look in their current directory.

```
sha = subprocess.check_output(
        ["git", "rev-parse", "HEAD"],
        cwd="src/some-project").decode("ascii").strip()
```

This will get the current git hash of the project, assuming the project is a git directory. If it is not, git rev-parse HEAD will return non-zero and cause an exception to be raised.

Note that we had to decode the output, since subprocess.check_output, like most functions in subprocess, returns a *byte string*, not a Unicode string. In this case, rev-parse HEAD always returns a hexadecimal string, so we used the ascii codec. This will fail on any non-ASCII characters.

There are some circumstances under which using the high-level abstractions are impossible. For example, having to send standard input or read output in chunks is not possible with them.

Popen runs a subprocess and allows fine-grained control on the inputs and outputs. While all things are, indeed, possible, most things are not easy to do correctly. The shell pattern of writing long pipelines is both unpleasant to implement; even more unpleasant to make sure there are no lingering deadlock conditions; and, most of all, unnecessary.

If a short message into standard input is needed, the best way is to use the method communicate.

```
proc = Popen(["docker", "login", "--password-stdin"], stdin=PIPE)
out, err = proc.communicate(my_password + "\n")
```

If longer input is needed, having the `communicate` buffer it all in memory might be problematic. While it is possible to write to the process in chunks, doing it without potentially getting deadlocks is nontrivial. The best option is often to use a temporary file:

```
with tempfile.TemporaryFile() as fp:
    fp.write(contents)
    fp.write(of)
    fp.write(email)
    fp.flush()
    fp.seek(0)
    proc = Popen(["sendmail"], stdin=fp)
    result = proc.poll()
```

In fact, in this case, we can even use the `check_call` function:

```
with tempfile.TemporaryFile() as fp:
    fp.write(contents)
    fp.write(of)
    fp.write(email)
    fp.flush()
    fp.seek(0)
    check_call(["sendmail"], stdin=fp)
```

If you are used to running processes in shell, you are probably used to long pipelines:

```
$ ls -l | sort | head -3 | awk '{print $3}'
```

As noted above, it is a best practice in Python to avoid true command parallelism: in all of the cases, we tried to finish one stage before reading from the next. In Python, in general, using `subprocess` is only used for calling out to external commands. For preprocessing of inputs, and post-processing of outputs, we usually use Python's built-in processing abilities: in the case above, we would use `sorted` slices and string manipulation to simulate the logic.

The commands for text and number processing are seldom useful in Python, which has a good in-memory model for doing such processing. The general case for calling commands in scripts is for things that manipulate data in a way that is either only documented as accessible by commands – for example, querying processes via `ps -ef`, or where the alternative to the command is a subtle library, sometimes requiring binary binding, such as in the case of `docker` or `git`.

This is one place where translating shell scripts into Python must be done with care and thought. Where the original had a long pipeline that depended on ad hoc string manipulation via awk or sed, Python code can be less parallel and more obvious. It is important to note that in those cases, there is something lost in translation: the original low-memory requirements and transparent parallelism. However, in return we get more maintainable and more debuggable code.

4.3 Networking

Python has plenty of networking support. It has it from the lowest level: support of the socket-based system calls to high-level protocol supports. Some of the best approaches for problems are with built-in libraries. For other problems, the best solution is a third-party library.

The most straightforward translation of low-level networking APIs is in the socket module. This module exposes the socket object.

The HTTP protocol is simple enough so we can implement a simple client straight from the Python interactive command prompt.

```
>>> import socket, json, pprint
>>> s = socket.socket()
>>> s.connect(('httpbin.org', 80))
>>> s.send(b'GET /get HTTP/1.0\r\nHost: httpbin.org\r\n\r\n')
40
>>> res = s.recv(1024)
>>> pprint.pprint(json.loads(
...               res.decode('ascii').split('\r\n\r\n', 1)[1]))
{'args': {},
 'headers': {'Connection': 'close', 'Host': 'httpbin.org'},
 'origin': '73.162.254.113',
 'url': 'http://httpbin.org/get'}
```

The line `s = socket.socket()` creates a new socket *object*. There are various things that we can do with socket objects. One of them is to connect them to an endpoint: in this case, to the server httpbin.org, port 80. The default socket type is a *stream*, *internet* type: this is the way UNIX refers to TCP sockets.

After the socket is connected, we can send bytes to it. Note – on sockets, only byte strings can be sent. We read back the result and do some ad hoc HTTP response parsing – and parse the actual content as JSON.

While in general, it is better to use a real HTTP client, this showcases how to write low-level socket code. This can be useful, for example, if we want to diagnose a problem by replaying exact messages.

The `socket` API is subtle, and the above example has a few incorrect assumptions in it. In most cases, this code will work but will fail in strange ways in the face of corner cases.

The `send` method is allowed to not send all the data, if not all of it can fit into the internal kernel-level send buffer. This means that it can do a "partial send." It returned `40`, above, which was the entire length of the byte string. Correct code would have checked for the return value and send the remaining chunks until nothing is left. Luckily, Python already has a method to do it: `sendall`.

However, a more subtle problem occurs with `recv`. It will return as much as the kernel-level buffer has, because it does not know how much the other side intended to send. Again, much of the time, especially for short messages, this will work fine. For protocols like HTTP 1.0, the correct behavior is to read until the connection is closed.

Here is a fixed version of the code:

```
>>> import socket, json, pprint
>>> s = socket.socket()
>>> s.connect(('httpbin.org', 80))
>>> s.sendall(b'GET /get HTTP/1.0\r\nHost: httpbin.org\r\n\r\n')
>>> resp = b''
>>> while True:
...     more = s.recv(1024)
...     if more == b'':
...             break
...     resp += more
...
>>> pprint.pprint(json.loads(resp.decode('ascii').split('\r\n\r\n')[1]))
{'args': {},
 'headers': {'Connection': 'close', 'Host': 'httpbin.org'},
 'origin': '73.162.254.113',
 'url': 'http://httpbin.org/get'}
```

This is a common problem in networking code, and one that can happen using higher-level abstractions as well. Things can appear to work in simple cases while failing to work in more extreme circumstances, such as high-load or network congestion.

There are ways to test for these things. One of them is using proxies that exhibit extreme behaviors. Writing, or customizing those, will require low-level network coding using `socket`.

Python also has higher-level abstractions for networking. While the `urllib` and `urllib2` modules are part of the standard library, best practices on the web evolve fast, and in general, for higher-level abstractions, third-party libraries are usually better.

One of the most popular is a third-party library, `requests`. With `requests`, getting a simple HTTP page is much simpler.

```
>>> import requests, pprint
>>> res=requests.get('http://httpbin.org/get')
>>> pprint.pprint(res.json())
{'args': {},
 'headers': {'Accept': '*/*',
             'Accept-Encoding': 'gzip, deflate',
             'Connection': 'close',
             'Host': 'httpbin.org',
             'User-Agent': 'python-requests/2.19.1'},
 'origin': '73.162.254.113',
 'url': 'http://httpbin.org/get'}
```

Instead of crafting our own HTTP requests out of raw bytes, all we needed to do was to give a URL, similar to a URL we might type into a browser. Requests parsed it to find the host to connect to (`httpbin.org`) the port (80, the default for HTTP) and the path (`/get`). Once the response came in, it automatically parsed it into headers and content, and it allowed us to access the content directly as JSON.

As easy as `requests` is to use, however, it is almost better to put in a little bit more effort and use the `Session` object. Otherwise, the default session is used. This leads to code with nonlocal side effects: one sub-library that calls requests changes some session state, which leads to another sub-library's calls to act differently. For example, HTTP cookies are shared across a session.

The code above would be better written as:

```
>>> import requests, pprint
>>> session = requests.Session()
>>> res = session.get('http://httpbin.org/get')
>>> pprint.pprint(res.json())
{'args': {},
 'headers': {'Accept': '*/*',
             'Accept-Encoding': 'gzip, deflate',
             'Connection': 'close',
             'Host': 'httpbin.org',
             'User-Agent': 'python-requests/2.19.1'},
 'origin': '73.162.254.113',
 'url': 'http://httpbin.org/get'}
```

In this example, the request is simple and session state does not matter. However, this is a good habit to get into: even in the interactive interpreter, to avoid using the `get`, `put`, and other functions directly and using only the session interface.

It is natural to use an interactive environment to prototype code, which would later make it into a production program. By keeping good habits like this, we ease the transition.

4.4 Summary

Python is a powerful tool for automating operating system operations. This comes from a combination of having libraries that are thin wrappers around native operating system calls and powerful third-party libraries.

This allows us to get close to the operating systems, without any intervening abstractions, as well as to write high-level code that does not care about the details when these do not matter.

This combination often makes Python a superior alternative for writing scripts, instead of using the UNIX shell. It does require a different way of thinking: Python is not as suitable for the long pipeline of text transformers approach, but in practice, those long pipelines of text transformers turn out to be an artifact of shell limitations.

With a modern memory-managed language, it is often easier to read the entire text stream into memory, and then manipulate it without being limited to only those transformations that can be specified as pipes.

CHAPTER 5

Testing

It is too often the case that code used for automating systems does not have the same attention for testing as application code. DevOps teams are often small and under tight deadlines. Such code is also hard to test, since it is meant to automate large systems, and proper isolation for testing is nontrivial.

However, testing is one of the best ways to increase code quality. It helps make code more maintainable in many ways. It also lowers defect rates. For code where defects can often mean total system outage, since it often touches *all* the parts of the system, this is important.

5.1 Unit Testing

Unit tests serve several distinct purposes. It is important to keep these purposes in mind, as the resulting pressures on the unit tests are sometimes at odds.

The first purpose is as an API usage example. This is sometimes summarized with the somewhat-inaccurate term "test driven development," and sometimes summarized with another other somewhat-inaccurate term, "the unit tests are the documentation."

Test driven development means writing the unit tests before the logic, but it usually has little effect on the final source code commit, which contains both the unit tests and the logic, unless care is taken to preserve the original branch-wise commit history.

However, what does show up in the commit is the "unit tests as ways to exercise the API." It is, ideally, *not* the only documentation of the API. However, it does serve as a useful reference of last resort: at the very least, we know that the unit tests are calling the API correctly and get back the results they expect.

Another reason is to gain confidence that the logic expressed in the code does the right thing. Again, this is often referred to with the misnomer "regression tests," after the most common such test: a test to make sure that a bug, detected by someone, is truly fixed. However, since the developer of the code is aware of the potential edge cases and

M. Zadka, *DevOps in Python*, https://doi.org/10.1007/978-1-4842-4433-3_5

trickier flows, they are often in a position to add that test *before* such a bug makes it out into the externally-observed code change: however, such a confidence-increasing test looks exactly like a "regression test."

A final reason is to avoid incorrect future changes. This is different from the "regression test," above, in that often the case being tested is straightforward for the code as is, and the flows involved are already covered by other tests. However, it seems like some potential optimizations or other natural changes might break this case, so including it helps a future maintenance programmer.

When writing a test, it is important to think about which of those goals it is meant to accomplish.

A good test will accomplish more than one. All tests have two potential impacts:

- Make the code better by helping future maintenance work.
- Make the code worse by making future maintenance work harder.

Every test will do some of both. A good test does more of the first, and a bad test does more of the second. One way to reduce the bad impact is to consider the question: "Is this test testing something that the code promises to do?" If the answer is "no," this means it is valid to change the code in some way that will break the test, but will not cause any bugs. This means the test has to be changed or discarded.

When writing tests, as much as possible, it is important to test the actual contract of the code.

Here is an example:

```
def write_numbers(fout):
    fout.write("1\n")
    fout.write("2\n")
    fout.write("3\n")
```

This function writes a few numbers into a file.

A bad test might look like this:

```
class DummyFile:

    def __init__(self):
        self.written = []

    def write(self, thing):
        self.written.append(thing)
```

```
def test_write_numbers():
    fout = DummyFile()
    write_numbers(fout)
    assert_that(fout.written, is_(["1\n", "2\n", "3\n"]))
```

The reason this is a bad test is because it checks for a promise that `write_numbers` never made: that each write only writes one line.

A future refactor might look like this:

```
def write_numbers(fout):
    fout.write("1\n2\n3\n")
```

This would keep the code *correct* – all users of `write_numbers` would still have correct files – but would cause a change in the test.

A slightly more sophisticated approach would be to concatenate the strings written.

```
class DummyFile:

    def __init__(self):
        self.written = []

    def write(self, thing):
        self.written.append(thing)

def test_write_numbers():
    fout = DummyFile()
    write_numbers(fout)
    assert_that("".join(fout.written), is_("1\n2\n3\n"))
```

Note that this test would work both before and after the hypothetical "optimization" that we suggested above. However, this still tests more than the implied contract of `write_numbers`. After all, the function is supposed to operate on files: it might use another method to write.

The test above would break if we modified `write_numbers` to:

```
def write_numbers(fout):
    fout.writelines(["1\n",
                     "2\n",
                     "3\n"]
```

A good test is one that would only break if there was a bug in the code. However, this code still works for the users of `write_numbers`, meaning that the maintenance now involved unbreaking a test, pure overhead.

Since the contract is to be able to write to file objects, it is best to supply a file object. In this case, Python has one ready-made:

```
def test_write_numbers():
    fout = io.StringIO()
    write_numbers(fout)
    assert_that(fout.getvalue(), is_("1\n2\n3\n"))
```

In some cases, this will require writing a custom fake. We will cover the concept of fakes, and how to write them, later.

We talked about the *implicit* contract of `write_numbers`. Since it had no documentation, we could not know what the original programmer's intent was. This is, unfortunately, common – especially in internal code, only used by other pieces of the project. Of course, it is better to clearly document programmer intent. In the face of lack of clear documentation, however, it is important to make *reasonable assumptions* on the implicit contract.

Above, we used the functions `assert_that` and `is_` to verify that the values were what we expected. Those functions come from the `hamcrest` library. This library, ported from Java, allows specifying properties of structures and checks that they are satisfied.

When using the `pytest` test runner to run unit tests, it is possible to use regular Python operators with the `assert` keyword and get useful test failures. However, this binds the tests to a specific runner, as well as having a specific set of assertions that get treated especially for useful error messages.

Hamcrest is an open-ended library: while it has built-in assertions for the usual things (equality, comparisons, sequence operations, and more), it also allows defining specific assertions. Those come in handy when handing complicated data structures, such as those returned from APIs, or when only specific assertions can be guaranteed by the contract (for example, the first three characters can be arbitrary but must be repeated somewhere inside of the rest of the string).

This allows to test the *exact* contract of the function. In particular, this is another tool in avoiding testing "too much": testing implementation details that can change, requiring changing the test when no real users have been broken. This is crucial for three reasons.

One is straightforward: time spent updating tests that could have been avoided is time wasted. DevOps teams are usually small, and there is little room to waste resources.

The second is that getting used to changing tests when they fail is a bad habit. It means when behavior has changed as a result of a *bug*, people will assume that the right thing to do is to update the test.

Finally, and most importantly, the combination of those two will lower the return on investment on unit testing, and even worse, the *perceived* return on investment. As a result, there will be organizational pressure to spend less time writing tests. Bad tests that test implementation details are the single biggest cause for the meme that "it is not worth it to write unit tests for DevOps code."

As an example, let's assume we have a function where all we can assert confidently is that the result has to be divisible by one of the arguments.

```
class DivisibleBy(hamcrest.core.base_matcher.BaseMatcher):

    def __init__(self, factor):
        self.factor = factor

    def _matches(self, item):
        return (item % self.factor) == 0

    def describe_to(self, description):
        description.append_text('number divisible by')
        description.append_text(repr(self.factor))

def divisible_by(num):
    return DivisibleBy(num)
```

By convention, we wrap constructors in a function. This is usually useful if we want to convert the argument to a matcher, which in this case would not make sense.

```
def test_scale():
    result = scale_one(3, 7)
    assert_that(result,
                any_of(divisible_by(3),
                       divisible_by(7)))
```

We would get an error like the following:

```
Expected: (number divisible by 3 or number divisible by 7)
     but: was <17>
```

It lets us test exactly what the contract of `scale_one` promises: in this case, that it would scale up one of the arguments by an integer factor.

The emphasis on the importance of testing precise contracts is not accidental. This emphasis, which is a skill that is possible to learn and has principles that are possible to teach, makes unit tests into something that accelerate the process of writing code rather than make it slower.

Much of the reason people have aversion to unit tests as "something that wastes time for DevOps engineers" and leads to a lot of poorly tested code that is foundational for business processes such as deployment of software is this misconception. Properly applying principles of high-quality unit testing leads to a more reliable foundation for operational code.

5.2 Mocks, Stubs, and Fakes

Typical DevOps code has outsized effects on the operating environment. Indeed, this is almost the definition of good DevOps code: it replaces a significant amount of manual work. This means that *testing* DevOps code needs to be done carefully: we cannot simply spin up a few hundred virtual machines for each test run.

Automating operations means writing code that, run haphazardly, can have significant impact on production systems. When testing the code, it is worthwhile to have as few of these side effects as possible. Even if we have high-quality staging systems, sacrificing one every time there is a bug in operational code would lead to a lot of wasted time. It is important to remember that unit tests run on the *worst* code produced: the act of running them, and fixing bugs, means even code committed into feature branches is likelier to be in better condition.

Because of that, we often try to run unit tests against a "fake" system. Classifying what we mean by "fake," and how it impacts both unit tests and code design, is important: it is worthwhile thinking about how to test the code well before starting to write it.

The neutral term for things that substitute for the systems not under test is "test doubles." Fakes, mocks, and stubs usually have more precise meaning, though in casual conversation they will be used interchangeably.

The most "authentic" test double is a "verified fake." A verified fake fully implements the interface of the system not under test, though often simplified: perhaps less efficiently implemented, often without touching any external operating system. The "verified" refers to the fact that the fake has its *own* tests, verifying that it does indeed implement the interface.

An example of a verified fake is using a memory-only SQLite database instead of a file-based one in tests. Since SQLite has its own tests, this is a verified fake: we can be confident it behaves like a real SQLite database.

Below the verified fake is the fake. The fake implements an interface but often does it in such a rudimentary form that the implementation is simple, and not worth the effort to test.

For example, it is possible to create an object with the same interface as `subprocess.Popen` but that never actually runs the process: instead, it simulates a process that consumes all standard input, and outputs some predetermined content into standard output and exits with a predetermined code.

This object, if simple enough, might be a *stub*. A stub is a simple object that answers with predetermined data, always the same, holding almost no logic. This makes it easier to write, but it does make it constrained in what tests it can do.

An *inspector*, or a *spy*, is an object that attaches to a test double and monitors the calls. Often, part of the contract of a function is that it will call some method with specific values. An inspector records the calls and can be used in assertions to make sure the right calls got the right arguments.

If we combine an inspect or with a stub or a fake, we get a *mock*. Since this means that the stub/fake will have more functionality than the original (at least, whatever is needed to check the recording), this can lead to some side effects. However, the simplicity and immediacy of creating mocks often compensates by making testing code simpler.

5.3 Testing Files

The filesystem is, in many ways, the most important thing about a UNIX system. While the slogan "everything is a file" falls short of describing modern systems, the filesystem is still at the heart of most operations.

The filesystem has several properties that are worthwhile to consider when thinking about testing file-manipulation code.

First, filesystems tend to be robust. While bugs in filesystems are not unknown, they are rare, far between, and usually only triggered by extreme conditions or an unlikely combination of conditions.

Next, filesystems tend to be fast. Consider the fact that unpacking a source tarball, a routine operation, will create many small files (on the order of several kilobytes) in quick succession. This is a combination of fast system-call mechanisms combined with sophisticated cache semantics when reading or writing files.

Filesystems also have a curious fractal property: with the exception of some esoteric operations, a sub-sub-sub-directory supports the same semantics as the root directory.

Finally, filesystems have a very thick interface. Some of it will be built into Python, even – consider that the module system reads files directly. There are also third-party C libraries that will use their own internal wrappers to access the filesystem as well as several ways to open files even in Python: the built-in `file` object as well as the `os.open` low-level operations.

This combines to the following conclusion: for most file-manipulation code, faking out or mocking the filesystem is a low return on investment. The investment, in order to make sure we are only testing the contract of a function, is considerable; since the function could, conceivably, switch to low-level file-manipulation operations, we would need to reimplement a significant portion of Unix file semantics. The return is low; using the filesystem directly is fast, reliable, and, as long as the code merely allows us to pass an alternative "root path," almost side-effect free.

The best way to design file-manipulation code is to allow passing in such a "root path" argument, even if the default is `/`. Given such design, the best way to test is to create a temporary directory, populate it appropriately, call the code, and then garbage collect the directory.

If we create the temporary directory using Python's built-in `tempfile` module, then we can configure the Tox runner to put the temporary file inside of Tox's built-in temporary directory, thus keeping the general file system clean and, usually, being compatible with whatever version control `ignore` file already ignores Tox artifacts.

```
setenv =
    TMPDIR = {envtmpdir}
commands =
    python -m 'import os;os.makedirs(sys.argv[1])' {envtmpdir}
    # rest of test commands
```

Creating the temporary directory is important, since Python's `tempfile` will only use the environment variable if pointing to a real directory.

As an example, we will write tests for a function that looks for `.js` files and renames them to `.py`.

```
def javascript_to_python_1(dirname):
    for fname in os.listdir(dirname):
        if fname.endswith('.js'):
            os.rename(fname, fname[:3] + '.py')
```

This function uses the `os.listdir` call to find the file names and then renames them with `os.rename`.

```
def javascript_to_python_2(dirname):
    for fname in glob.glob(os.path.join(dirname, "*.js")):
        os.rename(fname, fname[:3] + '.py')
```

This function uses the `glob.glob` function to filter by wildcard all the files that match the `*.js` pattern.

```
def javascript_to_python_3(dirname):
    for path in pathlib.Path(dirname).iterdir():
        if path.suffix == '.js':
            path.rename(path.parent.joinpath(path.stem + '.py'))
```

The function uses the built-in module `pathlib` (new in Python 3) to iterate on the directory and find its children.

The real function under test is not sure which implementation to use:

```
def javascript_to_python(dirname):
    return random.choice([javascript_to_python_1,
                          javascript_to_python_2,
                          javascript_to_python_3])(dirname)
```

Since we cannot be sure which *implementation* the function will use, we are left with only one choice: test the actual contract.

In order to write a test, we will define some helper code. This code, in a real project, will live in a dedicated module, possibly named something like `helpers_for_tests`. This module would be *tested*, with its own unit tests.

We first create a context manager for our temporary directory. This will ensure, as much as ensuring is possible, that the temporary directory will be cleaned up.

```
@contextlib.contextmanager
def get_temp_dir():
    temp_dir = tempfile.mkdtemp()
    try:
        yield temp_dir
    finally:
        shutil.rmtree(temp_dir)
```

Since this test needs to create a lot of files, and we do not care about their contents too much, we define a helper method for that.

```
def touch(fname, content=''):
    with open(fname, 'a') as fpin:
        fpin.write(content)
```

Now with the help of these functions, we can finally write a test:

```
def test_javascript_to_python_simple():
    with get_temp_dir() as temp_dir:
        touch(os.path.join(temp_dir, 'foo.js'))
        touch(os.path.join(temp_dir, 'bar.py'))
        touch(os.path.join(temp_dir, 'baz.txt'))
        javascript_to_python(temp_dir)
        assert_that(set(os.listdir(temp_dir)),
                    is_({'foo.py', 'bar.py', 'baz.txt'}))
```

For a real project, we would write more tests, many of them possibly using our `get_temp_dir` and `touch` helpers above.

If we have a function that is supposed to check a specific path, we can have it take an argument to "relativize" its paths.

For example, let us say we want a function to analyze our Debian installation paths and give us a list of all domains that we download packages from.

```
def _analyze_debian_paths_from_file(fpin):
    for line in fpin:
        line = line.strip()
```

```
        if not line:
            continue
        line = line.split('#', 1)[0]
        parts = line.split()
        if parts[0] != 'deb':
            continue
        if parts[1][0] == '[':
            del parts[1]
        parsed = hyperlink.URL.from_text(parts[1].decode('ascii'))
        yield parsed.host
```

A naive approach would be to test `_analyze_debian_paths_from_file`. However, it is an internal function and has *no* contract. The implementation can change, perhaps reading the files and then scanning all strings, or possibly breaking up this function and letting the top-level handle the line loop.

Instead, we want to test the public API:

```
def analyze_debian_paths():
    for fname in os.listdir('/etc/apt/sources.list.d'):
        with open(os.path.join('/etc/apt/sources.list.d', fname)) as fpin:
            yield from _analyze_debian_paths_from_file(fpin)
```

However, we cannot control the directory `/etc/apt/sources.list.d` without root privileges, and even with root privileges, this would be a risk: letting each test run control such a sensitive directory. Additionally, many Continuous Integration systems are not designed for running tests with root privileges, for good reasons, making this a problematic approach.

Instead, we can generalize the function a little bit. This means *intentionally* expanding the *official, public* API of the function to allow testing. This is definitely a trade-off.

However, the expansion is minimal: all we need is an explicit directory in which to work. In return, we get to simplify our testing requirements while avoiding any kind of "patching," which inevitably starts poking at private implementation details.

```
def analyze_debian_paths(relative_to='/'):
    sources_dir = os.path.join(relative_to, 'etc/apt/sources.list.d')
    for fname in os.listdir(sources_dir):
```

```
        with open(os.path.join(sources_dir, fname)) as fpin:
            yield from _analyze_debian_paths_from_file(fpin)
```

Now, using the same helpers as before, we can write a simple test for this:

```
def test_analyze_debian_paths():
    with get_temp_dir() as root:
        touch(os.path.join(root, 'foo.list'),
              content='deb http://foo.example.com\n')
        ret = list(analyze_debian_paths(relative_to=root))
        assert(ret, equals_to(['foo.example.com']))
```

Again, in a real project, we would write more than one test and try to make sure many more cases are covered. Those could be built using the same techniques.

It is a good habit to add a `relative_to` parameter to any function that accesses specific paths.

5.4 Testing Processes

Testing process-manipulation code is often a subtle endeavor, full of trade-offs. In theory, process running code has a thick interface with the operating system; we covered the `subprocess` module, but it is possible to use the `os.spawn*` functions directly, or even use code `os.fork` and `os.exec*` functions. Likewise, the standard output/input communication mechanism can be implemented in many ways, including using the `Popen` abstraction or directly manipulating file descriptors with `os.pipe` and `os.dup`.

Process-manipulation code can also be some of the most fragile. Running external commands depends on the behavior of those commands, as a starting point. The inter-process communication means that the flow is inherently concurrent. It is too easy to make the mistake of making the tests rely on ordering assumptions that are not always true. Those mistakes can lead to "flaky" tests: ones that pass most of the time, but fail under seemingly random circumstances.

Those ordering assumptions can sometimes be true more often on development machines, or unloaded machines, which means bugs will only be exposed in production, or possibly in production only in extreme circumstances.

This is one of the reasons the chapter about using processes concentrated on ways to reduce concurrency and have things more sequential. For this reason, too, it is worthwhile, carefully designing process code to be reliably testable. That design, in itself, will often cause pressure on the code to be simple and reliable.

If the code just uses `subprocess.check_call` and `subprocess.check_output`, without taking advantage of exotic parameters, we can often use a simplified form of a pattern called "dependency injection" to make it testable. In this case, "dependency injection" is just a fancy way of saying "passing parameters to a function."

Consider the following function:

```
def error_lines(container_name):
    logs = subprocess.check_output(["docker", "logs", container_name])
    for line in logs:
        if 'error' in line:
            return line
```

This function is unpleasant to test. We can use advanced patching to replace `subprocess.check_output`, but this would be error prone and rely on implementation details. Instead, we can explicitly elevate that implementation detail into being a part of the contract:

```
def error_lines(container_name, runner=subprocess.check_output):
    logs = runner(["docker", "logs", container_name])
    for line in logs:
        if 'error' in line:
            yield line.strip()
```

Now that `runner` is part of the official interface, testing becomes much easier. This might seem a trivial change, but it is deeper than it looks; in some sense, `error_lines` has now voluntarily constrained its interface to process running.

We might want to test it with something like the following:

```
def test_error_lines():
    container_name = 'foo'

    def runner(args):
        if args[0] != 'docker':
            raise ValueError("Can only run docker", args)
```

```
        if args[1] != 'logs':
            raise ValueError("Can only run docker logs", args)
        if args[2] != container_name:
            raise ValueError("No such container", args[2])
        return iter(["hello\n", "error: 5 is not 6\n", "goodbye\n"])
    ret = error_lines(container_name, runner=runner)
    assert_that(list(ret), is_(["error: 5 is not 6"))
```

Note that, in this case, we did not restrict ourselves to *only* checking the contract: `error_lines` could have run, for example, `docker logs -- <container_name>`. However, one advantage of our method is that we can slowly improve our fidelity and *only* improve the test.

For example, we can add to runner:

```
def runner(args):
    if args[0] != 'docker':
        raise ValueError("Can only run docker", args)
    if args[1] != 'logs':
        raise ValueError("Can only run docker logs", args)
    if args[2] == '--':
        arg_container_name = args[3]
    else:
        arg_container_name = args[2]
    if args_container_name != container_name:
        raise ValueError("No such container", args[2])
    return iter(["hello\n", "error: 5 is not 6\n", "goodbye\n"])
```

This will *still* work with the old version of the code and will also work with post-modification code. Fully emulating the docker is not realistic or worthwhile. However, this approach would slowly improve the accuracy of the test, with no downsides.

If a significant amount of our code interfaces, for example, with docker, we can eventually factor out a mini-docker-emulator like that into its own test helper library.

Using higher-level abstractions for process running helps with this sort of approach. The `seashore` library, for example, separates the part that calculates the commands from the low-level runner, which allows substituting only the low-level one.

```
def error_lines(container_name, executor):
    logs, _ignored = executor.docker.logs(container_name).batch()
    for line in logs.splitlines():
        if 'error' in line:
            yield line.strip()
```

When run in production, somewhere at the top, an executor object will be created with code that looks like this:

```
executor = seashore.Executor(seashore.Shell())
```

That object will be passed down to whatever is calling error_lines and used there. In general, when using seashore, we leave the creation of the executor to the top-level functionality.

In the test, we create our own shell:

```
@attr.s
class DummyShell:

    _container_name = attr.ib()

    def batch(self, *args, **kwargs):
        if (args == ['docker', 'logs', self._container_name] and
            kwargs == {}):
            return "hello\nerror: 5 is not 6\ngoodbye\n", ""
        raise ValueError("unknown command", self, args, kwargs)

def test_error_lines():
    container_name = 'foo'
    executor = seashore.Executor(DummyShell(container_name))
    ret = error_lines(container_name, executor)
    assert_that(list(ret), is_(["error: 5 is not 6"]))
```

Using the attrs library, especially when writing various fakes, is often a good idea. Fakes tend to be, intentionally, simple objects. Since they will be involved in assertions and exceptions, it is useful to have high-quality representations of them. This is exactly the kind of boilerplate that attrs helps reduce.

Again, we might need to slowly upgrade our fidelity.

Because processes are so hard to test, it is good to use process running only when necessary. Especially when porting over shell scripts to Python – often a good idea when they grow in complexity – it is good to substitute long pipelines with in-memory data processing.

Especially if we factor the code the right way, with the data processing as a simple pure function that takes an argument and returns a value, the bulk of the code becomes a pleasure to test.

Imagine, for example, the pipeline,

```
ps aux | grep conky | grep -v grep | awk '{print $2}' | xargs kill
```

This will kill all processes that have `conky` in their names.

Here is a way to refactor the code to make it easier to test:

```
def get_pids(lines):
    for line in lines:
        if 'conky' not in line:
            continue
        parts = line.split()
        pid_part = parts[1]
        pid = int(pid_part)
        yield pid

def ps_aux(runner=subprocess.check_output):
    return runner(["ps", "aux"])

def kill(pids, killer=os.kill):
    for pid in pids:
        killer(pid, signal.SIGTERM)

def main():
    kill(get_pid(ps_aux()))
```

Note how the most complicated code is now in a pure function: `get_pids`. Hopefully, this means most bugs will be there, and we can unit test against them.

The code that is harder to unit test, `get_pids`, where we have to do ad hoc dependency injection, is now in simple functions that have fewer failure modes.

The main logic is in functions that do data processing. Testing those just requires supplying simple data structure and observing the return value. *Moving* potential bugs

from the system-related code, which requires more effort to unit test, to the *pure logic*, which is easier to unit test, means *reducing* the bugs; more bugs will be caught with unit tests.

5.5 Testing Networking

In the `requests` library documentation, using the `Session` object falls under the "advanced" section. This is unfortunate. For anything other than throwaway scripts, or interactive REPL usage, using the `Session` object is the best option. Testing is by far the least of the reasons – but once `Session` is used, testing becomes a lot easier.

Simple example code using `requests` might look like this:

```
def get_files(gist_id):
    gist = requests.get(f"https://api.github.com/gists/{gist_id}").json()
    result = {}
    for name, details in gist["files"].items():
        result[name] = requests.get(details["raw_url"]).content
    return result
```

This would be hard to test in isolation. Instead, we rewrite it to take an explicit session object:

```
def get_files(gist_id, session=None):
    if session is None:
        session = requests.Session()
    gist = session.get(f"https://api.github.com/gists/{gist_id}").json()
    result = {}
    for name, details in gist["files"].items():
        result[name] = session.get(details["raw_url"]).content
    return result
```

The code is almost identical. However, now testing becomes a simple matter of writing an object with a `get` method.

```
@attr.s(frozen=True)
class Gist:
    files = attr.ib()
```

```
@attr.s(frozen=True):
class Response:

    content = attr.ib()

    def json(self):
        return json.loads(content)

@attr.s(frozen=True)
class FakeSession:

    _gists = attr.ib()

    def get(self, url):
        parsed = hyperlink.URL.from_text(url)
        if parsed.host == 'api.github.com':
            tail = path.rsplit('/', 1)[-1]
            gist = self._gists[tail]
            res = dict(files={name: f'http://example.com/{tail}/{name}'
                              for name in gist.files})
            return Repsonse(json.dumps(res))
        if parsed.host == 'example.com':
            _ignored, gist, name = path.split('/')
            return Response(self.gists[gist][name])
```

This is a bit long-winded. We can sometimes, if this functionality is localized and writing a whole helper library is not worth it, use the `unittest.mock` library.

```
def make_mock():
    gist_name = 'some_name'
    files = {'some_file': 'some_content'}
    session = mock.Mock()
    session.get.content.return_value = 'some_content'
    session.get.json.return_value = json.dumps({'files': 'some_file'})
    return session
```

This is a "quick and dirty" hack, counting on the fact (that is *not* in the contract) that the file content is retrieved using `content`, and the gist's logical structure is retrieved using `json`. However, it is often better to write a quick test using mocks that depend a little on the implementation details rather than not writing a test at all.

It is important to think of tests like this as "technical debt" and improve them at some point to depend more on the contract and less on the implementation details. A good way to do it is to put a comment in the code, and link it to an issue tracker. This also makes it obvious to test code readers that this is still a work in progress.

The other important thing is that, if a new implementation breaks the test, the right way to fix it is, in general, *not* to write another test against the new implementation. The right way to fix it is to move more of the test to contract-based testing. This can be done by *first* improving the test, but making sure it runs against the old code. *Then* comes refactoring the code and seeing the test still passing.

When writing network code that deals with lower-level concepts, such as sockets, similar ideas still apply. Since the creation of the socket object is separate from any usage of it, a lot of mileage can be gotten out of writing functions that accept socket objects, and creating them outside.

In order to simulate extreme conditions and see if our code can work in spite of them, we might want to use something like the following as a socket fake:

```
@attr.s
class FakeSimpleSocket:

    _chunk_size = attr.ib()

    _received = attr.ib(init=False, factory=list)

    _to_send = attr.ib()

    def connect(self, addr):
        pass

    def send(self, blob):
        actually_sent = blob[:chunk_size]
        self._received.append(actually_sent)
        return len(actually_sent)

    def recv(self, max_size):
        chunk_size = min(max_size, self._chunk_size)
        received, self._to_send = (self._to_send[:chunk_size],
                                   self._to_send[chunk_size:])
        return received
```

This allows us to control the size of "chunks." An extreme test would be to use a chunk_size of 1. This means bytes would go out one at a time, and they would be received one at a time. No real network would be this bad, but a unit test allows us to simulate more extreme conditions than any reasonable network.

This fake is useful to test networking code. For example, this code does some ad hoc HTTP to get a result:

```
def get_get(sock):
    sock.connect(('httpbin.org', 80))
    sock.send(b'GET /get HTTP/1.0\r\nHost: httpbin.org\r\n\r\n')
    res = sock.recv(1024)
    return json.loads(res.decode('ascii').split('\r\n\r\n', 1)[1]))
```

It has a subtle bug in it. We can uncover the bug with a simple unit test, using the socket fake.

```
def test_get_get():
    result = dict(url='http://httpbin.org/get')
    headers = 'HTTP/1.0 200 OK\r\nContent-Type: application/json\r\n\r\n'
    output = headers + json.dumps(result)
    fake_sock = FakeSimpleSocket(to_send=output, chunk_size=1)
    value = get_get(fake_sock)
    assert_that(value, is_(result))
```

This test would fail: our get_get assumes a good quality network connection, and this simulates a bad one. It would succeed if we changed chunk_size to 1024.

We could run the test in a loop, testing chunk sizes from 1 to 1024. In a real test we would also check the sent data, and possibly also send invalid results to see the response. The important thing, however, is that none of those things need setting up clients or servers, or trying to realistically simulate bad networks.

5.6 Summary

Teams rely on DevOps code to keep systems functional and observable. The correctness of DevOps code is critical. Writing proper tests will help improve code correctness. Taking proper test-writing principles into account will help reduce the burden of modifying tests when making correct changes to the code.

CHAPTER 6

Text Manipulation

Automation of UNIX-based systems often involves text manipulation. Many programs are configured with textual configuration files. Text is the output format, and the input format, of many systems. While tools like `sed`, `grep`, and `awk` have their place, Python is a powerful tool for sophisticated text manipulation.

6.1 Bytes, Strings, and Unicode

When manipulating text or text-like streams, it is easy to write code that fails in funny ways when encountering a foreign name, or emoji. These are no longer mere theoretical concerns: you will have users from the entire world, who insist on their usernames reflecting how they spell their names. You will have people who write `git` commits with emojis in them. In order to make sure to write robust code, which does not fail in ways that, to be fair, seem a lot *less* funny when they case a 3 a.m. page, it is important to understand that "text" is a subtle thing.

You can understand the distinction, or you can wake up at 3 a.m. when someone tries to log in with an emoji username.

Python 3 has two distinct types that both represent the kind of things that are often in UNIX "text" files: bytes and strings. Bytes correspond to what RFCs usually refer to as an "octet-stream." This is a sequence of values that fit into 8 bits, or in other words, a sequence of numbers that are in the range 0 to 256 (including 0 and not including 256). When all of these values are below 128, we call the sequence "ASCII" (American Standard Code of Information Interchange) and assign to the numbers the meaning ASCII has assigned them. When all of these values are between 32 and 128 (including 32 and not including 128), we call the sequence "printable ASCII," or "ASCII text." The first 32 characters are sometimes called "Control characters." The "Ctrl" key on keyboards is a reference to that – its original purpose was to be able to input those characters.

M. Zadka, *DevOps in Python*, https://doi.org/10.1007/978-1-4842-4433-3_6

ASCII only encompasses the English alphabet, used in "America." In order to represent text in (almost) any language, we have Unicode. Unicode code points are (some of the) numbers between `0` and `2**32` (including `0` and not including `2**32`). Each Unicode code point is assigned a meaning. Successive versions of the standards leave assigned meanings as is, but add meanings to more numbers. An example is the addition of more emojis. The International Standards Organization, ISO, ratifies versions of Unicode in its 10464 standards. For this reason, Unicode is sometimes called ISO-10464.

Unicode points that are also ASCII have the same meaning – if ASCII assigns a number "uppercase A," then so does Unicode.

Properly speaking, only Unicode is "text." This is what Python strings represent. Converting bytes to strings, or vice versa, is done with an *encoding*. The most popular encoding these days is UTF-8. Confusingly, turning the bytes *to* text is "decoding." Turning the text to bytes is "encoding."

Remembering the difference between encoding and decoding is crucial in order to manipulate textual data. A way to remember it is that since UTF-8 *is* an encoding, moving from strings *to* UTF-8 encoded data is "encoding," while moving from UTF-8 encoded data to strings is "decoding."

UTF-8 has an interesting property: when given a Unicode string that happens to be ASCII, it will produce bytes with the values of the code points. This means that "visually," the encoded and decoded form will look the same.

```
>>> "hello".encode("utf-8")
b'hello'
>>> "hello".encode("utf-16")
b'\xff\xfeh\x00e\x00l\x00l\x00o\x00'
```

We show the example with UTF-16 to show that this is not a trivial property of encodings. Another property of UTF-8 is that if the bytes are *not* ASCII, and UTF-8 decoding of the bytes succeeds, then it is unlikely that they were encoded with a different encoding. This is because UTF-8 was designed to be *self-synchronizing*: starting at a random byte, it is possible to synchronize with the string with a limited number of bytes being checked. Self-synchronization was designed to allow recovery from truncation and corruption, but as a side benefit, it allows *detecting* invalid characters reliably, and thus detect if the string was UTF-8 to begin with.

This means “try decoding with UTF-8” is a safe operation; it will do the right thing for ASCII-only texts, and it will, of course, work for UTF-8 encoded texts and will fail cleanly for things that are neither ASCII nor UTF-8 encoded – either text in a different encoding or a binary format such as JPEG.

For Python, fails cleanly means “throws an exception.”

```
>>> snowman = '\N{snowman}'
>>> snowman.encode('utf-16').decode('utf-8')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xff in position 0:
invalid start byte
```

For random data, this will also tend to fail:

```
>>> struct.pack('B'*12,
                *(random.randrange(0, 256)
                for i in range(12))
 ).decode('utf-8')
```

The errors are random, since the inputs are random. Some example errors might be:

```
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xe2 in position 4:
invalid continuation byte
UnicodeDecodeError: 'utf-8' codec can't decode byte 0x98 in position 2:
invalid start byte
```

It is a good exercise to try and run this a few times; it will almost never succeed.

6.2 Strings

The Python string object is subtle. From one perspective it appears to be a sequence of characters: and a *character* is a string of length 1.

```
>>> a="hello"
>>> for i, x in enumerate(a):
...     print(i, x, len(x))
...
```

```
0 h 1
1 e 1
2 l 1
3 l 1
4 o 1
```

The string “hello” has five elements, each of which is a string of length 1. Since the string is a sequence, the usual sequence operations work on it.

We can create a slice by specifying both endpoints:

```
>>> a[2:4]
'll'
```

or just the end:

```
>>> a[:2]
'he'
```

or just the beginning:

```
>>> a[3:]
'lo'
```

We can also use negative indices to count from the end:

```
>>> a[:-3]
'he'
```

And of course, we can reverse a string by specifying an extended slice with a negative step:

```
>>> a[::-1]
'olleh'
```

However, strings also have quite a few methods that are *not* part of the general sequence interface and are useful when analyzing text.

The `startswith` and `endswith` methods are useful, since text analysis is often around the ends.

```
>>> "hello world".endswith("world")
True
```

A little-known feature is that endswith allows a tuple of strings and will check if it ends with any of these strings:

```
>>> "hello world".endswith(("universe", "world"))
True
```

An example where it comes in useful is testing for a few common endings:

```
>>> filename.endswith((".tgz", ".tar.gz"))
```

We can easily test here whether a file has either of the common suffixes for a gzipped tarball: either the tgz or tar.gz suffix.

The strip and split methods are useful for parsing the kind of ad hoc formats that many UNIX files or utilities come in. For example, the file /etc/fstab contains static mounts.

```
with open("/etc/fstab") as fpin:
    for line in fpin:
        line = line.rstrip('\n')
        line = line.split('#', 1)[0]
        if not line:
            continue
        device, path, fstype, options, freq, passno = line.split()
        print(f"Mounting {device} on {path}")
```

This parses the file and prints a summary. The first line in the loop strips out the newline. The rstrip method strips from the right (the end) of the string.

Note that rstrip, as well as strip, accept a sequence of characters to remove. This means that passing a *string* to rstrip means "any of the characters in the string" and *not* "remove occurrences of this string." This does not affect one-character arguments to rstrip, but it does mean that longer strings are almost always a mistaken use.

We then remove comments, if any. We skip empty lines. Any line that is not empty, we use the split with no argument, to split on any sequence of whitespaces. Conveniently, this convention is common to several formats, and the correct handling is built into the specification of split.

Lastly, we use a *format* string to format the output for easy consumption.

This is a typical usage of string parsing, and it is the kind of code that replaces long pipelines in shell.

Finally, the join method on a string uses it as a "glue" and glues together an iterable of strings.

The simple example of ' '.join(["hello", "world"]) will return "hello world," but this is only scratching the surface of join. Since it accepts an iterable, we have the ability to pass it anything that supports iteration.

```
>>> names=dict(hello=1,world=2)
>>> ' '.join(names)
'hello world'
```

Since iterating on a dictionary objects yields the list of keys, passing it to join means that we get a string with the list of keys, joined together.

We can also pass in a generator:

```
>>> '-*-'.join(str(x) for x in range(3))
'0-*-1-*-2'
```

This allows calculating sequences on the fly and joining them, without the need to have intermediate storage for the sequence.

The usual question about join is why it is a method on the "glue" string rather than a method on sequences. The reason is exactly this: we can pass in any iterable, and the glue string will glue in the bits in it.

Note that join does nothing to single-element iterables:

```
>>> '-*-'.join(str(x) for x in range(0))
'0'
```

6.3 Regular Expressions

Regular expressions are a special DSL for specifying properties of strings, also called "patterns." They are common in many utilities, although each implementation will have its own idiosyncrasies. In Python, regular expressions are implemented by the re module. It fundamentally allows two modes of interaction – one where regular expressions are auto-parsed at the time of text analysis, and one where they are parsed in advance.

In general, the latter style is preferred. Auto-parsing the regular expression is suited only to an interactive loop, where they will be used quickly and forgotten. For this reason, we will not really cover this usage here.

In order to *compile* a regular expression, we use `re.compile`. This function returns a regular expression object that will look for strings that match the expression. The object can be used to do several things: for example, find one match, find all matches, or even replace the matches.

The regular expression mini-language has a lot of subtlety. Here, we will cover only the basics that we need to illustrate how to use regular expressions effectively.

Most characters stand in for themselves. The regular expression `hello`, for example, matches exactly `hello`. The `.` stands in for any character. So `hell.` would match `hello` and `hella`, but not `hell` – since the latter does not have any character corresponding to the `.`. Square brackets delimit "character classes": for example, `wom[ae]n` matches both `women` and `woman`. Character classes can also have ranges in them – `[0-9]` matches any digit, `[a-z]` matches any lowercase character, and `[0-9a-fA-F]` matches any hexadecimal digit (hexadecimal digits and numbers pop up a lot in many places, since two hexadecimal digits correspond exactly to a standard byte).

We also have the "repeat modifiers" that modify the expression that precedes them. For example, `ba?b` matches both `bb` and `bab` – the `?` stands for "zero or one." The * stands for any number: so `ba*b` stands for `bb`, `bab`, `baab`, `baaab`, and so on. `If` we want "at least one," `ba+b` will match almost everything that `ba*b` matches, except for `bb`. Finally, we have the exact counters: `ba{3}b` matches `baaab` while `ba{1,2}b` matches `bab` and `baab` and nothing else.

In order to make a special character (like `.` or *) match itself, we prefix it with a backslash. Since in Python strings, backslash has other meanings, Python supports "raw" strings. While we can use any string to denote a regular expression, often raw strings are easier.

For example, we want a DOS-like filename regular expression: `r"[^.]{1,8}\.[^.]{0,3}."` This will match, say, `readme.txt` but not `archive.tar.gz`. Note that to match a literal . we escaped it with a backslash. Also note that we used an interesting character class: `[^.]`. This means "anything except `.`": the ^ means "exclude" inside of a characer class.

Regular expressions also support *grouping*. Grouping does two things: it allows addressing parts of the expression, and it allows treating a part of the expression as a single object in order to apply one of the repeat operations to it. If only the latter is needed, this is a "non-capture" group, denoted by `(?:....)`.

For example, `(?:[a-z]{2,5}-){1,4}[0-9]` will match `hello-3` or `hello-world-5` but not `a-hello-2` (since the first part is not two characters long) or `hello-world-this-is-too-long-7` since it is made up of six repetitions of the inner pattern, and we specified a maximum of four.

This allows arbitrary nesting; for example `(?:(?:[a-z]{2,5}-){1,4}[0-9];)+` allows any semicolon-terminated, separated sequence of the previous pattern: for example `az-2;hello-world-5;` will match but `this-is-3;not-good-match-6` will not, since it is missing the `;` at the end.

This is a good example of how complex regular expressions can get. It is easy to use this dense mini-language inside Python to specify constraints on strings that are hard to understand.

Once we have a regular expression object, there are two main methods on it: `match` and `search`. The `match` method will look for matches at the beginning of the string, while `search` will look for the first match, wherever it may start. When they find a match, they return a match object.

```
>>> reobj = re.compile('ab+a')
>>> m = reobj.search('hello abba world')
>>> m
<_sre.SRE_Match object; span=(6, 10), match='abba'>
>>> m.group()
'abba'
```

The first method that is often used is `.group()`, which returns the part of the string matched. This method can get a part of the match, if the regular expression contained *capturing* groups. A capturing group is usually marked with `()`.

```
>>> reobj = re.compile('(a)(b+)(a)')
>>> m = reobj.search('hello abba world')
>>> m.group()
'abba'
>>> m.group(1)
'a'
>>> m.group(2)
'bb'
>>> m.group(3)
'a'
```

When the number of groups is significant, or when modifying the group, managing the indices to the group can prove to be a challenge. If analysis of the groups is needed, we can also *name* the groups.

```
>>> reobj = re.compile('(?P<prefix>a)(?P<body>b+)(?P<suffix>a)')
>>> m = reobj.search('hello abba world')
>>> m.group('prefix')
'a'
>>> m.group('body')
'bb'
>>> m.group('suffix')
'a'
```

Since regular expressions can get dense, there is a way to make them a bit easier to read: the verbose mode.

```
>>> reobj = re.compile(r"""
... (?P<prefix>a) # The beginning -- always an a
... (?P<body>b+)  # The middle -- any numbers of b, for emphasis
... (?P<suffix>a) # An a at the end to properly anchor
... """, re.VERBOSE)
>>> m = reobj.search("hello abba world")
>>> m.groups()
('a', 'bb', 'a')
>>> m.group('prefix'), m.group('body'), m.group('suffix')
('a', 'bb', 'a')
```

When compiling regular expressions with the flag `re.VERBOSE`, whitespace is ignored, and comments, Python-like:

to end of line, are also ignored. In order to match a space or #, they need to be backslash escaped.

This allows writing long regular expressions while still making them easier to understand with judicious line breaks, spaces, and comments.

Regular expressions are loosely based on the mathematical theory of finite automaton. While they *do* go beyond the constraints of what finite automata can match, they are not fully general. Among other things, they are poorly suited for *nested* patterns; whether matching parentheses or HTML elements, they are not a good fit for regular expressions.

6.4 JSON

JSON is a hierarchical file format that has the advantage of being simple to parse, and reasonably easy to read and write by hand. It has its origins on the web: the name stands for "JavaScript Object Notation." Indeed, it is still popular on the internet; one reason to care about JSON is that many web APIs use JSON as a transfer format.

It is also useful, however, in other places. For example, in JavaScript projects, `package.json` includes the dependencies of this project. Parsing this is often useful to determine third-party dependencies for security or compliance audits, for example.

In theory, JSON is a format defined in *Unicode*, not *bytes*. When serializing, it takes a data structure and transforms it into a Unicode string, and when deserializing, it takes a Unicode string and returns a data structure. Recently, however, the standard was amended to specify a preferred encoding: `utf-8`. With this addition, now the format is also defined as a byte stream.

However, note that in some use cases, the encoding is still separate from the format. In particular, when sending or receiving JSON over HTTP, the HTTP encoding is the ultimate truth. Even then, though, when no encoding is explicitly specified, UTF-8 should be assumed.

JSON is a simple serialization format, only supporting a few types:

- Strings
- Numbers
- Booleans
- A `null` type
- Arrays of JSON values
- "Objects": dictionaries mapping strings to JSON values

Note that JSON does not full specify numerical ranges or precision. If precise integers are required, usually the range `-2**53` to `2**53` can be assumed to be representable precisely.

Although the Python `json` library has the ability to read/write directly to files, in practice we almost always separate the tasks; we read as much data as we need and pass the string directly to JSON.

The two functions that are the most important in the `json` module are `loads` and `dumps`. The `s` at the end stands for "string," which is what those functions accept and return.

```
>>> thing = [{"hello": 1, "world": 2}, None, True]
>>> json.dumps(thing)
'[{"hello": 1, "world": 2}, null, true]'
>>> json.loads(_)
[{'hello': 1, 'world': 2}, None, True]
```

The `None` object in Python maps to the JSON `null` object, booleans in Python map to booleans in JSON, and numbers and strings map to number and strings. Note that the Python JSON parsing libraries makes ad hoc decisions about whether a number should map to an integer or a float based on the notation it uses:

```
>>> json.loads("1")
1
>>> json.loads("1.0")
1.0
```

It is important to remember not all JSON loading libraries make the same decision, and in some cases, this can lead to interoperability problems.

For debugging reasons, it is often useful to be able to "pretty print" JSON. The `dumps` function can do that, with some extra arguments. The usual set of arguments for pretty printing is the following:

```
json.dumps(thing, sort_keys=True, indent=4)
```

If we want to round-trip into an equivalent, but pretty version, we can even do this:

```
json.dumps(json.loads(encoded_string), sort_keys=True, indent=4)
```

Finally, at the command line, the module:*json.tool* will do this automatically:

```
$ python -m json.tool < somefile.json | less
```

This is an easy way to scan through dumped JSON and look for interesting information.

Note that with Python 3.7 and above, `sort_keys` should be used judiciously; since all dictionaries are ordered by insertion, *not* using `sort_keys` will keep the original order in the dictionary.

One frequently missed type from JSON is a date-time type. Usually this is represented with strings, and is the most common need for a "schema" to parse JSON against, in order to know which strings to convert to a `datetime` object.

6.5 CSV

The CSV format has a few advantages. It is constrained: it always represents scalar types in a two-dimensional array. For this reason, there are not a lot of surprises that can go in. In addition, it is a format that imports natively into spreadsheet applications like Microsoft Excel or Google Sheets. This comes in handy when preparing reports.

Examples of such reports are of breaking down expenses for paying for third-party services for the financial department, or a report on incidents managed and time to recovery for management. In all these cases, having a format that is easy to produce and import into spreadsheet applications allows for easy automation of the task.

Writing CSV files is done with `csv.writer`. A typical example involves serializing a homogenous array, an array of things with the same type.

```
@attr.s(frozen=True, auto_attribs=True)
class LoginAttempt:
    username: str
    time_stamp: int
    success: bool
```

This class represents a login attempt by some user, at a given time, and with a record of the success of the attempt. For a security audit, we need to send the auditors an Excel file of the login attempts.

```
def write_attempts(attempts, fname):
    with open(fname, 'w') as fpout:
        writer = csv.writer(fpout)
        writer.writerow(['Username', 'Timestamp', 'Success'])
        for attempt in attempts:
            writer.writerow([
```

```
                attempt.username,
                attempt.time_stamp,
                str(attempt.success),
            ])
```

Note that by convention, the first row should be a "title row." Though the Python API does not enforce it, it is highly recommended to follow this convention. In this example, we first wrote a "title row" with the names of the fields.

Then we looped through the attempts. Note that CSV can only represent strings and numbers, so instead of relying on thinly documented standards on how a boolean will be written out, we have done so explicitly.

This way, if the auditor asks for that field to be "yes/no," we can change our explicit serialization step to match.

When it comes to reading CSV files, there are two main approaches.

Using `csv.reader` will return an iterator that yields parsed row by parsed row, as a list. However, assuming the convention about the first row being the names of fields has been followed, `csv.DictReader` will yield nothing for the first row, and a *dictionary* for every subsequent row, using field names as keys. This enables more robust parsing in the face of end users adding fields or changing their order.

```
>>> reader = csv.DictReader(fileobj)
>>> list(reader)
[OrderedDict([('Username', 'alice'),
              ('Timestamp', '1514793600.0'),
              ('Success', 'False')]),
 OrderedDict([('Username', 'bob'),
              ('Timestamp', '1539154800.0'),
              ('Success', 'True')])]
```

Reading the same CSV that we have written in the previous example will yield reasonable results. The dictionary maps the field names to the values. It is important to note that the types have all been forgotten, and everything is returned as a string. Unfortunately, CSV does not keep type information.

It is sometimes tempting to just "improvise" parsing CSV files with `.split`. However, CSV has quite a few corner cases that are not readily apparent.

For example,

```
1,"Miami, FL","he""llo"
```

is properly parsed as

```
('1', 'Miami, FL', 'he"llo')
```

For the same reason, it is a good idea to avoid writing CSV files using anything other than `csv.writer`.

6.6 Summary

Much of the content that is needed for many DevOps tasks arrives as text: logs, JSON dumps of data structures, or a CSV file of paid licenses. Understanding what “text” is and how to manipulate it in Python allow much of the automation that is the cornerstone of DevOps, be it through build automation, monitoring result analysis, or just preparing summaries for easy consumption by others.

CHAPTER 7

Requests

Many systems expose a web-based API. Automating web-based APIs is easy with the `requests` library. It is designed to be easy to use while still exposing a lot of powerful features. Using `requests` is almost always better than using Python's standard library HTTP client facilities.

7.1 Sessions

As mentioned before, it is better to work with explicit sessions in `requests`. It is important to remember that there is no such thing as working *without* a session in requests; when working with the "functions," it is using the global session objects.

This is problematic for several reasons. For one, this is exactly the kind of "global mutable shared state" that can cause it to be hard to diagnose bugs. For example, when connecting to a website that uses cookies, another user of requests connecting to the same website could override the cookies. This leads to subtle interactions between potentially far-apart pieces of code.

The other reason it is problematic is because this makes code nontrivial to unittest. The `request.get`/`request.post` functions would have to be explicitly mocked, instead of supplying a fake `Session` object.

Last but not least, some functionality is only accessible when using an explicit `Session` object. If the requirement to use it comes later, for example, because we want to add a tracing header or a custom user-agent to all requests, refactoring all code to use explicit sessions can be subtle.

It is much better, for any code that has any expectation to be long lived, to use an explicit session object. For similar reasons, it is even better to make most of this code not construct its own `Session` object, but rather get it as an argument.

M. Zadka, *DevOps in Python*, https://doi.org/10.1007/978-1-4842-4433-3_7

This allows initializing the session elsewhere, closer to the main code. This is useful because this means that decisions about which proxies to use, and when, can happen closer to the end-user requirements rather than in abstract library code.

A session object is constructed with `requests.Session()`. After that, the only interaction should be with the object. The session object has all the HTTP methods: `s.get`, `s.put`, `s.post`, `s.patch`, and `s.options`.

Sessions can be used as contexts:

```
with requests.Session() as s:
    s.get(...)
```

At the end of the context, all pending connections will be cleaned up. This can sometimes be important, especially if a web server has strict usage limits that we cannot afford to exceed for any reason.

Note that counting on Python's reference counting to close the connections can be dangerous. Not only is that not guaranteed by the language (and will not be true, for example, in PyPy), but small things can easily prevent this from happening. For example, the session can be captured as a local variable in a stack trace, and that stack trace can be involved in a circular data structure. This means that the connections will not get closed for a potentially long time: not until Python does a circular garbage collection cycle.

The session supports a few variables that we can mutate in order to send all requests in a specific way. The most common one to have to edit is `s.auth`. We will touch more about the authentication capabilities of `requests` later.

Another variable that is useful to mutate is `session.headers`. Those are the default headers that are sent with every request. This can sometimes be useful for the `User-Agent` variable. Especially when using `requests` for testing our own web APIs, it is useful to have an identifying string in the agent. This will allow us to check the server logs and distinguish which requests came from tests as opposed to real users.

```
session.headers = {'User-Agent': 'Python/MySoftware ' + __version__ }
```

This will allow checking which version of the test code caused a problem. Especially if the test code crashes the server, and we want to disable it, this can be invaluable in diagnosis.

The session also holds a `CookieJar` in the `cookies` member. This is useful if we want to explicitly flush, or check, cookies. We can also use it to persist cookies to disk and recover them, if we want to have restartable sessions.

We can either mutate the cookie jar or replace it wholesale: any `cookielib.CookieJar`-compatible object can work.

Finally, the session can have a client-side certificate in it, for use in situations where this kind of authentication is desired. This can either be a `pem` file (the key and the certificate concatenated) or a tuple with the paths to the certificate and key file.

7.2 REST

The REST name stands for "Representational State Transfer." It is a loose, and loosely applied, standard of representing information on the web. It is often used to map a row-oriented database structure almost directly to the web, allowing an edit operation; when used this way, it is often also called the "CRUD" model: Create, Retrieve, Update, and Delete.

When using REST for CRUD, a few web operations are frequently used.

The first is create maps to `POST`, which is accessed via `session.post`. In some sense, although the first on the list, it is the least "RESTful" of the four. This is because its semantics are not "replay" safe.

This means that if the `session.post` raises a network-level error, for example, `socket.error`, it is not obvious how to proceed; was the object actually created? If one of the fields in the object must be unique, for example, an e-mail address for a user, then replaying is safe: it will fail if the creation operation succeeded earlier.

However, this depends on *application* semantics, which means that it is not possible to replay generically.

Luckily, the HTTP methods typically used for the other operations *are* "replay safe." This property is also known as idempotency, inspired by (though not identical with) the mathematical notion of idempotent functions. This means that if a network failure occurred, sending the operation again is safe.

All operations that follow, if the server follows correct HTTP semantics, are replay safe.

The Update operation is usually implemented with `PUT` (for a whole-object update) or `PATCH` (when changing specific fields).

The Delete operation is implemented with HTTP `DELETE`. The replay safety here is subtle; whether a replay succeeds or fails with an "object not found," at the end we are left in a known state.

Retrieve, implemented with HTTP `GET`, is almost always a read-only operation, and so is replay safe: it is safe to retry after a network failure.

Most REST services, nowadays, use JSON as the state representation. The `requests` library has special support for JSON.

```
>>> pprint(s.get("https://httpbin.org/json").json())
{'slideshow': {'author': 'Yours Truly',
               'date': 'date of publication',
              'slides': [{'title': 'Wake up to WonderWidgets!', 'type': 'all'},
                         {'items': ['Why <em>WonderWidgets</em> are great',
                                    'Who <em>buys</em> WonderWidgets'],
                           'title': 'Overview',
                           'type': 'all'}],
               'title': 'Sample Slide Show'}}
```

The return value from a request, `Response`, has a `.json()` method that assumes the return value is JSON and parses it. While this only saves one step, it is a useful step to save in a multistage process where we get some JSON-encoded response only to use it in a further request.

It is also possible to auto-encode the request body as JSON:

```
>>> resp = s.put("https://httpbin.org/put", json=dict(hello=5,world=2))
>>> resp.json()['json']
{'hello': 5, 'world': 2}
```

The combination of those two, with a multistep process, is often useful.

```
>>> res = s.get("https://api.github.com/repos/python/cpython/pulls")
>>> commits_url = res.json()[0]['commits_url']
>>> commits = s.get(commits_url).json()
>>> print(commits[0]['commit']['message'])
```

This example of getting a commit message from the first pull request on the CPython project is a typical example of using a good REST API. A good REST API includes *URLs* as resource identifiers. We can pass those URLs to a further request to get more information.

7.3 Security

The HTTP security model relies on *certification authorities*, often shortened to "CAs." Certification authorities cryptographically sign public keys as belonging to a specific domain (or, less commonly, IP). In order to enable key rotation and revocation, certificate authorities do not sign the public key with their *root key* (the one trusted by the browser). Rather, they sign a "signing key," which signs the public key. These "chains," where each key signs the next one, until the ultimate key is the one the server is using, can get long: often there is a three- or four-level deep chain.

Since certificates sign the *domain*, and often domains are co-hosted on the same IP, the protocol that requests the certificate includes "Server Name Indication," or SNI. SNI sends the server name, unencrypted, which the client wants to connect to. Then the server responds with the appropriate certificate, and proves that it owns the private key corresponding to the signed public key using cryptography.

Finally, optionally the client can engage in a cryptographic proof of its own identities. This is done through the slightly misnamed "client-side certificates." The client side has to be initialized with both a certificate *and* a private key. Then the client sends the certificate, and if the server trusts the certifying authority, proves that it owns the corresponding private key.

Client-side certificates are seldom used in browsers but can be sometimes used by programs. For a program, they are usually *easier* secrets to deploy: most clients, `requests` included, support reading them out of files already. This makes it possible to deploy them using systems that make secrets available via files, like Kubernetes. It also means it is easier to manage permissions on them via normal UNIX system permissions.

Note that usually, client-side certificates are not owned by a public CA. Rather, the server owner operates a *local* CA, which through some locally determined procedure, signs certificates for clients. This can be anything from an IT person signing manually, to a Single-Sign On portal that auto-signs certificates.

In order to authenticate *server-side* certificates, `requests` needs to have a source of client-side root CAs in order to be able to successfully accomplish secure connections. Depending on subtleties of the `ssl` build process, it might or might not have access to the *system* certificate store.

The best way to make sure to have a good set of root CAs is to install the package `certifi`. This package has Mozilla-compatible certificates, and `requests` will use it natively.

This is useful when making connections to the internet; almost all sites are tested to work with Firefox, and so have a compatible certificate chain. If the certificate fails to validate, the error `CERTIFICATE VALIDATE FAILED` is thrown. There is a lot of unfortunate advice on the internet, including in `requests` documentation, about the "solution" of passing in the flag `verify=False`. While there are rare cases where this flag would make sense, it almost never does. Its usage violates the core assumption of TLS: that the connection is encrypted and tamper-proof.

For example, having a `verify=False` on the request means that any cookies or authentication credentials can now be intercepted by anyone with the ability to modify in-stream packets. This is unfortunately common: ISPs and open access points often have operators with nefarious motivation.

A better alternative is to make sure that the correct certificates exist on the file system, and passing the path to the `verify` argument via `verify='/full/path'`. At the very least, this allows us a form of "trust on first use": manually get the certificate from the service, and bake it into the code. It is even better to attempt some out-of-band verification, for example, by asking someone to log in to the server and verify the certificate.

Choosing what SSL versions to allow, or what ciphers to allow, is slightly more subtle. There are, again, few reasons to do it: `requests` is set up with good, secure, defaults. However, sometimes there are overriding concerns: for example, avoiding a specific SSL cipher for a regulatory reason.

The first important thing to know is that `requests` is a wrapper around the `urllib3` library. In order to change low-level parameters, we need to write a customized `HTTPAdapter` and set the session object we are using to use the custom adapter.

```
from requests.adapters import HTTPAdapter
from requests.packages.urllib3.poolmanager import PoolManager

class MyAdapter(HTTPAdapter):
    pass

s = requests.Session()
s.mount('https://', MyAdapter())
```

This, of course, has no business logic effect: the MyAdapter class is not different from the HTTPAdapter class. But now that we have the mechanics for custom adapters, we can change the SSL versions:

```
class MyAdapter(HTTPAdater)
    def init_poolmanager(self, connections, maxsize, block=False):
        self.poolmanager = PoolManager(num_pools=connections,
                                       maxsize=maxsize,
                                       block=block,
                                       ssl_version=ssl.PROTOCOL_TLS)
```

Much like the ssl_version, we can also fine-tune the list of ciphers, using the ciphers= keyword argument. This keyword argument should be a string that has :-separated names of ciphers.

Requests also supports so-called "client-side" certificates. Seldom used for user-to-service communication, but sometimes used in microservice architectures, client-side certificates identify the client using the same mechanism that servers identify themselves: using cryptographically signed proofs. The client needs a private key and a corresponding certificate. These certificates will often be signed by a private CA, which is part of the local infrastructure.

The certificate and the key can be concatenated into the same file, often called a "PEM" file. In that case, initializing the session to identify with it is done via:

```
s = requests.Session()
s.cert = "/path/to/pem/file.pem"
```

If the certificate and the private key are in separate files, they are given as a tuple:

```
s = requests.Session()
s.cert = ("/path/to/client.cert", "/path/to/client.key")
```

Such key files must be carefully managed; anyone who has read access to them can pretend to be the client.

7.4 Authentication

This will be the default authentication sent with requests. Included in `requests` itself, the most commonly used authentication is *basic auth*.

For basic auth, this argument can be just a tuple, `(username, password)`. However, a better practice is to use an `HTTPBasicAuth` instance. This documents the intent better, and is useful if we ever want to switch to other authentication forms.

There are also third-party packages that implement the authentication interface and supply custom auth classes. The interface is pretty straightforward: it expects the object to be callable and will call the object with the `Request` object. It is expected that the call will mutate the `Requests`, usually by adding headers.

The official documentation recommends subclassing `AuthBase`, which is just an object that implements a `__call__` that raises a `NotImplementedError`. There is little need for that.

For example, the following is useful as an object that will sign AWS requests with the V4 signing protocol.

The first thing we do is make the URL "canonical." Canonicalization is a first step in many signing protocols. Since often higher levels of the software will have already parsed the content by the time the signature checker gets to look at it, we convert the signed data into a standard form that uniquely corresponds to the parsed version.

The most subtle part is the query part. We parse it, and re-encode it, using the `urlparse` built-in library.

```
def canonical_query_string(query):
    if not query:
        return ""
    parsed = parse_qs(url.query, keep_blank_values=True)
    return "?" + urlencode(parsed, doseq=True)
```

We use this function in our URL canonicalization function:

```
def to_canonical_url(url):
    url = urlparse(raw_url)
    path = url.path or "/"
    query = canonical_query_string(url.query)
```

```
    return (url.scheme +
            "://" +
            url.netloc +
            path +
            querystring)
```

Here we make sure the path is canonical: we translate an empty path to /.

```
from botocore.auth import SigV4Auth
from botocore.awsrequest import AWSRequest

def sign(request, *, aws_session, region, service):
    aws_request = AWSRequest(
        method=request.method.upper(),
        url=to_canonical_url(request.url),
        data=request.body,
    )
    credentials = aws_session.get_credentials()
    SigV4Auth(credentials, service, region).add_auth(request)
    request.headers.update(**aws_request.headers.items())
```

We create a function that uses botocore, the AWS Python SDK, to sign a request. We do that by "faking" an AWSRequest object with the canonical URL and the same data, asking for a signature, and then grabbing the headers from the "faked" request.

We use this as follows:

```
requests_session = requests.Session()
requests_session.auth = functools.partial(sign,
    aws_session=boto3.Session(),
    region='us-east-1',
    service='es',
)
```

functools.partial is an easy way to get a simple callable from the original function. Note that in this case, the region and the service are part of the auth "object." A more sophisticated approach would be to infer the region and service from the request's URL and use that. This is beyond the scope of this simple example. However, this should

give a good idea about how custom authentication schemes work: we write code that modifies the request to have the right authentication headers, and then put it in as the `auth` property on the session.

7.5 Summary

Saying "HTTP is popular" feels like an understatement. It is everywhere: from user-accessible services, through web-facing APIs, and even internally in many microservice architectures.

`requests` helps with all of these: it can help be part of monitoring a user-accessible service for health, it can help us access APIs in programs to analyze the data, and it can help us debug internal services to understand what their state is.

It is a powerful library, with many ways to fine-tune it to send exactly the right requests, and get exactly the right functions.

CHAPTER 8

Cryptography

Cryptography is a necessary component in many parts of secure architecture. However, just adding cryptography to the code does not make it more secure; care must be given to such topics as secrets generation, secrets storage, and plain-text management. Properly designing secure software is a complicated matter, more so when cryptography is involved.

Designing for security is beyond our scope here: this chapter only teaches the basic tools that Python has for cryptography, and how to use them.

8.1 Fernet

The `cryptography` module supports the `fernet` cryptography standard. It is named after an Italian, not French, wine: the "t" is pronounced. A good approximation for the pronunciation is like "fair-net."

`fernet` works for *symmetric* cryptography. It does not support partial or streaming decryption: it expects to read in the whole ciphertext and to return the whole plain text. This makes it suitable for names, text documents, or even pictures. However, videos and disk images are a poor fit for Fernet.

The cryptographic parameters of Fernet were chosen by domain experts, who researched available encryption methods, as well as the known, best attacks against them. One advantage in using Fernet is that it avoids the need for you to become an expert yourself. However, for completeness, we note that the Fernet standard uses AES-128 in CBC padding with PKCS7, and HMAC using SHA256 for authentication.

The Fernet standard is also supported by Go, Ruby, and Erlang and so is sometimes suitable for data exchange with other languages. It was especially designed so that using it *insecurely* is harder than using it correctly.

```
>>> k = fernet.Fernet.generate_key()
>>> type(k)
<class 'bytes'>
```


M. Zadka, *DevOps in Python*, https://doi.org/10.1007/978-1-4842-4433-3_8

The key is a short string of bytes. Managing the key securely is important: cryptography is only as good as its keys. If it is kept in a file, for example, the file should have minimal permissions and ideally be hosted on an encrypted file system.

The `generate_key` class method takes care to generate the key securely, using an operating-system level source of random bytes. However, it is still vulnerable to operating-system level flaws: for example, when cloning virtual machines, care must be taken that when starting the clone, it refreshes the source of randomness. This is admittedly an esoteric case, and whatever virtualization system is being used should have documentation on how to refresh the randomness source in its virtual machines.

```
>>> frn = fernet.Fernet(k)
```

The fernet class is initialized with a key. It will make sure that the key is valid.

```
>>> encrypted = frn.encrypt(b"x marks the spot")
>>> encrypted[:10]
b'gAAAAABb1'
```

Encryption is simple. It takes a string of bytes and returns an encrypted string. Note that the encrypted string is *longer* than the source string. The reason is that it is also signed with the secret key. This means that tampering with the encrypted string is detectable, and the Fernet API handles that by refusing to decrypt the string. This means that the value gotten back from decryption is *trustworthy*; it was indeed encrypted by someone who had access to the secret key.

```
>>> frn.decrypt(encrypted)
b'x marks the spot'
```

Decryption is done in the same way as encryption. Fernet does contain a version marker, so if vulnerabilities in these are found, it is possible to move the standard to a different encryption and hashing system.

Fernet encryption always adds the current date to the signed, encrypted information. Because of this, it is possible to limit the *age* of a message before decrypting.

```
>>> frn.decrypt(encrypted, ttl=5)
```

This will fail if the encrypted information (sometimes referred to as the "token") is older than five seconds. This is useful to prevent *replay* attacks: one where a previous encrypted token was captured and replayed instead of a new valid token. For example, if the encrypted token has a list of usernames that are allowed some access, and is retrieved using a subvertible medium, a user who is no longer allowed in can substitute the older token.

Ensuring token freshness would mean that no such list would be decoded, and everybody would be denied – which is no worse than if the medium was tampered with *without* having a token that was previously valid.

This can also be used to ensure good secret rotation hygiene. By refusing to decrypt anything older than, say, a week, we make sure that if the secret rotation infrastructure broke, we would fail loudly instead of succeeding silently, and thus fix it.

In order to support seamless key rotation, the Fernet module also has a `MultiFernet` class. `MultiFernet` takes a list of secrets. It encrypts with the first secret but will try decrypting with any secret.

This means that if we add a new key to the end, first, it will not be used for encryption. After the addition to the end is synchronized, we can remove the first key. Now all encryptions will be done via the second key; and even those instances where it is not synchronized yet will have the decryption key available.

This two-step process is designed to have zero "invalid decryption" errors while still allowing key rotation, which is important as a precautionary measure – and a well-tested rotation procedure means that if keys are leaked, the rotation procedure can minimize the harm they do.

8.2 PyNaCl

PyNaCl is a library wrapping the `libsodium` C library. `libsodium` is a fork of Daniel J. Bernstein's `libnacl,` which is why `PyNaCl` is named that way. (NaCl, or Sodium Chloride, is the chemical formula for salt. The fork took the name of the first element.)

PyNaCl supports both symmetric and asymmetric encryption. However, since cryptography supports symmetric encryption with Fernet, the main use of PyNaCl is for asymmetric encryption.

The idea of asymmetric encryption is that there is a private and a public key. The public key can easily be calculated from the private key but not vice versa; that is, the "asymmetry" it refers to. The public key is published, while the private key must remain a secret.

There are, in general, two basic operations supported with public-key cryptography. We can encrypt with the public key, in a way that can only be decrypted with the private key. We can also *sign* with the private key, in a way that can be verified with the public key.

As we have discussed earlier, modern cryptographic practice places as much value on *authentication* as it does on *secrecy*. This is because if the media the secret is transmitted on is vulnerable to eavesdropping, it is often vulnerable to modification. Secret modification attacks have had enough impact on the field that a cryptographic system is not considered complete if it does not guarantee both authenticity and secrecy.

Because of that, `libsodium`, and by extension `PyNaCl`, do not support encryption without signing, or decryption without signature verification.

In order to generate a private key, we just use the class method:

```
>>> from nacl.public import PrivateKey
>>> k = PrivateKey.generate()
```

The type of `k` is `PrivateKey`. However, at some point, we will usually want to persist the private key.

```
>>> type(k.encode())
<class 'bytes'>
```

The `encode` method encodes the secret key as a stream of bytes.

```
>>> kk = PrivateKey(k.encode())
>>> kk == k
True
```

We can generate a private key from the byte stream, and it will be identical. This means we can again keep the private key in a way we decide is secure enough: a secret manager, for example.

In order to encrypt, we need a *public key*. Public keys can be generated from private keys.

```
>>> from nacl.public import PublicKey
>>> target = PrivateKey.generate()
>>> public_key = target.public_key
```

Of course, in a more realistic scenario, public keys need to be stored somewhere: in a file, in a database, or just sent via the network. For that, we need to convert the public key into bytes.

```
>>> encoded = public_key.encode()
>>> encoded[:4]
b'\xb91>\x95'
```

When we get the bytes, we can regenerate the public key. It is identical to the original public key.

```
>>> public_key_2 = PublicKey(key_bytes)
>>> public_key_2 == public_key
True
```

A PyNaCl `Box` represents pair of keys: the first private, the second public. The `Box` signs with the private key, then encrypts with the public key. Every message that we encrypt always gets signed.

```
>>> from nacl.public import PrivateKey, PublicKey, Box
>>> source = PrivateKey.generate()
>>> with open("target.pubkey", "rb") as fpin:
...     target_public_key = PublicKey(fpin.read())
>>> enc_box = Box(source, target_public_key)
>>> result = enc_box.encrypt(b"x marks the spot")
>>> result[:4]
b'\xe2\x1c0\xa4'
```

This one signs using the `source` private key and encrypts using the `target`'s public key.

When we decrypt, we need to build the inverse box. This happens on a different computer: one that has the `target` *private key* but only the source's *public key*.

```
>>> from nacl.public import PrivateKey, PublicKey, Box
>>> with open("source.pubkey", "rb") as fpin:
...     source_public_key = PublicKey(fpin.read())
>>> with open("target.private_key", "rb") as fpin:
...     target = PrivateKey(fpin.read())
```

```
>>> dec_box = Box(target, source_public_key)
>>> dec_box.decrypt(result)
b'x marks the spot'
```

The decryption box decrypts with `target` private key and verifies the signature using `source`'s public key. If the information has been tampered with, the decryption operation automatically fails. This means that it is impossible to access plain-text information that is not correctly signed.

Another piece functionality that is useful inside of PyNaCl is cryptographic signing. It is sometimes useful to sign *without* encryption: for example, we can make sure to only use approved binary files by signing them. This allows the permissions for *storing* the binary file to be loose, as long as we trust that the permissions on *keeping the signing key secure* are strong enough.

Signing also involves asymmetric cryptography. The private key is used to sign, and the public key is used to verify the signatures. This means that we can, for example, check the public key into source control, and avoid needing any further configuration of the verification part.

We first have to generate the private signing key. This is similar to generating a key for decryption.

```
>>> from nacl.signing import SigningKey
>>> key = SigningKey.generate()
```

We will usually need to store this key (securely) somewhere for repeated use. Again, it is worthwhile remembering that anyone who can access the signing key can sign whatever data they want. For this, we can use encoding:

```
>>> encoded = key.encode()
>>> type(encoded)
<class 'bytes'>
```

The key can be reconstructed from the encoded version. That produces an identical key.

```
>>> key_2 = SigningKey(encoded)
>>> key_2 == key
True
```

For verification, we need to have the verification key. Since this is asymmetric cryptography, the verification key can be calculated from the signing key, but not vice versa.

```
>>> verify_key = key.verify_key
```

We will usually need to store the verification key somewhere, so we need to be able to encode it as bytes.

```
>>> verify_encoded = verify_key.encode()
>>> verify_encoded[:4]
b'\x08\xb1\x9e\xf4'
```

We can reconstruct the verification key. That gives an identical key. Like all `...Key` classes, it supports a constructor that accepts an encoded key and returns a key object.

```
>>> from nacl.signing import VerifyKey
>>> verify_key_2 = VerifyKey(verify_encoded)
>>> verify_key == verify_key_2
True
```

When we sign a message, we get an interesting object back:

```
>>> message = b"The number you shall count is three"
>>> result = key.sign(message)
>>> result
b'\x1a\xd38[....'
```

It displays as bytes. But it is not bytes:

```
>>> type(result)
<class 'nacl.signing.SignedMessage'>
```

We can extract the message and the signature from it separately:

```
>>> result.message
b'The number you shall count is three'
>>> result.signature
b'\x1a\xd38[...'
```

This is useful in case we want to save the signature in a separate place. For example, if the original is in an object storage, mutating it might be undesirable for various reasons. In those cases, we can keep the signatures "on the side." Another reason is to maintain different signatures for different purposes, or to allow key rotation.

If we do want to write the whole signed message, it is best to explicitly convert the result to bytes.

```
>>> encoded = bytes(result)
```

The verification returns back the verified message. This is the best way to use signatures; this way, it is impossible for the code to handle an unverified message.

```
>>> verify_key.verify(encoded)
b'The number you shall count is three'
```

However, if it is necessary to read the object itself from somewhere else, and then pass it into the verifier, that is also easy to do.

```
>>> verify_key.verify(b'The number you shall count is three',
...                   result.signature)
b'The number you shall count is three'
```

Finally, we can just use the result object as is to verify.

```
>>> verify_key.verify(result)
b'The number you shall count is three'
```

8.3 Passlib

Secure storage of passwords is a delicate matter. The biggest reason it is so subtle is that it has to deal with people who do not use password best practices. If all passwords were strong, and people never reused passwords from site to site, password storage would be straightforward.

However, people usually choose passwords with little entropy (`123456` is still unreasonably popular, as well as `password`), they have a "standard password" that they use for all websites, and they are often vulnerable to phishing attacks and social engineering attacks where they divulge the password to an unauthorized third party.

Not all of these threats can be stopped by correctly storing passwords, but many of them can, at least, be mitigated and weakened.

The `passlib` library is written by people who are well versed in software security, and tries to, at least, eliminate the most obvious mistakes when saving passwords. Passwords are never saved in plain text – always hashed.

Note that hashing algorithms for passwords are optimized for different use cases than hashing algorithms used for other reasons: for example, one of the things they try to deny is brute-force source mapping attacks.

Passlib hashes passwords with the latest vetted algorithms optimized for password storage, and they intended to avoid any possibility of side-channel attacks. In addition, "salt" is always used for hashing the passwords.

Although `passlib` can be used without understanding these things, it is worthwhile to understand them in order to avoid mistakes while using `passlib.`

Hashing means taking the users' passwords and running it through a function that is reasonably easy to compute but hard to invert. This means that even if an attacker gets access to the password database, they cannot recover users' passwords and pretend to be them.

One way that the attacker can try to get the original passwords is to try all combinations of passwords they can come up with, hash them, and see if they are equal to a password. In order to avoid this, special algorithms are used that are computationally hard. This means that an attacker would have to use a lot of resources in order to try many passwords, so that even if, say, only a few million passwords are tried, it would take a long time to compare. Lastly, attackers can use something called "rainbow tables" to precompute many hashes of common passwords, and compare them all at once against a password database. In order to avoid that, passwords are "salted" before they are hashed: a random prefix (the "salt") is added, the password is hashed, and the salt is prefixed to the hash value. When the password is received from the user, the salt is retrieved from the beginning of the hash value, before hashing it to compare.

Doing all of this from scratch is hard and even harder to get it right. Getting it "right" does not just mean having users log in, but being resilient to the password database being stolen. Since there is no feedback about that aspect, it is best to use a well-tested library.

The library is storage agnostic: it does not care where the passwords are being stored. However, it does care that it is possible to update the hashed passwords. This way, hashed passwords can get updated to newer hashing schemes as the need arises. While `passlib` does support various low-level interfaces, it is best to use the high-level interface of the `CryptContext`. The name is misleading, since it does no encryption; it is a reference to vaguely similar (and largely deprecated) functionality built into Unix.

The first thing to do is to decide on a list of supported hashes. Note that not all of them have to be *good* hashes; if we have supported bad hashes in the past, they still have to be in the list. In this example, we choose `argon2` as our preferred hash but allow a few more options.

```
>>> hashes = ["argon2", "pbkdf2_sha256", "md5_crypt", "des_crypt"]
```

Note that `md5` and `des` have serious vulnerabilities and are not suitable to use in real applications. We added them because there might be old hashes using them. In contrast, even though `pbkdf2_sha256` is, probably, worse than `argon2`, there is no urgent need to update it. We want to mark `md5` and `des` as deprecated.

```
>>> deprecated = ["md5_crypt", "des_crypt"]
```

Finally, after having made the decisions, we build the crypto context:

```
>>> from passlib.context import CryptContext
>>> ctx = CryptContext(schemes=hashes, deprecated=deprecated)
```

It is possible to configure other details, such as the number of rounds. This is almost always unnecessary, as the defaults should be good enough.

Sometimes we will want to keep this information in some configuration (for example, an environment variable or a file) and load it; this way, we can update the list of hashes without modifying the code.

```
>>> serialized = ctx.to_string()
>>> new_ctx = CryptContext.from_string(serialized)
```

When saving the string, note that it does contain newlines; this might impact where it can be saved. If needed, it is always possible to convert it to base64.

On user creation or change password, we need to hash the password before storing it. This is done via the `hash` method on the context.

```
>>> res = ctx.hash("good password")
```

When logging in, the first step is to retrieve the hash from storage. After retrieving the hash, and having the users' passwords from the user interface, we need to check that they match, and possibly update the hash if it is using a deprecated protocol.

```
>>> ctx.verify_and_update("good password", res)
(True, None)
```

If the second element were true, we would need to update the hash with the result. In general, it is not a good idea to specify a specific hash algorithm, but to trust the context defaults. However, in order to showcase the update, we can force the context to hash with a weak algorithm.

```
>>> res = ctx.hash("good password", scheme="md5_crypt")
```

In that case, `verify_and_update` would let us know we should update the hash:

```
>>> ctx.verify_and_update("good password", res)
(True, '$5$...')
```

In that case, we would need to store the second element in our password hash storage.

8.4 TLS Certificates

Transport Layer Security (TLS) is a cryptographic way to protect data in transit. Since one potential attack is man-in-the-middle, it is important to be able to verify that the endpoints are correct. For this reason, the public keys are *signed* by Certificate Authorities. Sometimes, it is useful to have a local certificate authority.

One case where that can be useful is in microservice architectures, where verifying each service is the right one allows a more secure installation. Another case where that is useful is for putting together an internal test environment, where using real certificate authorities is sometimes not worth the effort; it is easy enough to install the local certificate authority as locally trusted and sign the relevant certificates with it.

Another place that this can be useful is in running tests. When running integration tests, we would like to set up a realistic integration environment. Ideally, some of these tests would check that; indeed, TLS is used rather than plain text. This is impossible to test if, for purposes of testing, we downgrade to plain-text communication. Indeed, the root cause of many production security breaches is that the code, inserted for testing, to

enable plain-text communication, was accidentally enabled (or possible to maliciously enable) in production; and furthermore, it was impossible to test that such bugs did not exist, because the testing environment *did* have plain-text communication.

For the same reason, allowing TLS connections without verification in the testing environment is dangerous. This means that the code has a non-verification flow, which can accidentally turn on, or maliciously be turned on, in production, and is impossible to prevent with testing.

Creating a certificate manually requires access to the `hazmat` layer in `cryptography`. This is so named because this is dangerous; we have to judiciously choose encryption algorithms and parameters, and the wrong choices can lead to insecure modes.

In order to perform cryptography, we need a "back end." This is because originally it was intended to support multiple back ends. This design is mostly deprecated, but we still need to create it and pass it around.

```
>>> from cryptography.hazmat.backends import default_backend
```

Finally, we are ready to generate our private key. For this example, we will use 2048 bits, which is considered "reasonably secure" as of 2019. A complete discussion of which sizes provide how much security is beyond the scope of this chapter.

```
>>> from cryptography.hazmat.primitives.asymmetric import rsa
>>> private_key = rsa.generate_private_key(
...     public_exponent=65537,
...     key_size=2048,
...     backend=default_backend()
... )
```

As always in asymmetric cryptography, it is possible (and fast) to calculate the public key from the private key.

```
>>> public_key = private_key.public_key()
```

This is important, since the certificate only refers to the *public* key. Since the private key is never shared, it is not worthwhile, and actively dangerous, to make any assertions about it.

The next step is to create a *certificate builder*. The certificate builder will be used to add "assertions" about the public key. In this case, we are going to finish by *self-signing* the certificate, since CA certificates are self-signed.

```
>>> from cryptography import x509
>>> builder = x509.CertificateBuilder()
```

We then add names. Some names are required, though it is not important to have specific contents in them.

```
>>> from cryptography.x509.oid import NameOID
>>> builder = builder.subject_name(x509.Name([
... x509.NameAttribute(NameOID.COMMON_NAME, 'Simple Test CA'),
... ]))
>>> builder = builder.issuer_name(x509.Name([
... x509.NameAttribute(NameOID.COMMON_NAME, 'Simple Test CA'),
... ]))
```

We need to decide a validity range. For this, it is useful to be able to have a "day" interval for easy calculation.

```
>>> import datetime
>>> one_day = datetime.timedelta(days=1)
```

We want to make the validity range start "slightly before now." This way, it will be valid for clocks with some amount of skew.

```
>>> today = datetime.date.today()
>>> yesterday = today - one_day
>>> builder = builder.not_valid_before(yesterday)
```

Since this certificate will be used for testing, we do not need to have it be valid for a long time. We will make it valid for 30 days.

```
>>> next_month = today + (30 * day)
>>> builder = builder.not_valid_after(next_month)
```

The serial number needs to uniquely identify the certificate. Since keeping enough information to remember which serial numbers we used is complicated, we choose a

different path: choosing a random serial number. The probability of having the same serial number chosen twice is extremely low.

```
>>> builder = builder.serial_number(x509.random_serial_number())
```

We then add the public key that we generated. This certificate is made of assertions *about* this public key.

```
>>> builder = builder.public_key(public_key)
```

Since this is a CA certificate, we need to mark it as a CA certificate.

```
>>> builder = builder.add_extension(
... x509.BasicConstraints(ca=True, path_length=None),
... critical=True)
```

Finally, after we have added all the assertions into the builder, we need to generate the hash and sign it.

```
>>> from cryptography.hazmat.primitives import hashes
>>> certificate = builder.sign(
...    private_key=private_key, algorithm=hashes.SHA256(),
...    backend=default_backend()
... )
```

This is it! We now have a private key, and a self-signed certificate that claims to be a CA. However, we will need to store them in files.

The PEM file format is friendly to simple concatenation. Indeed, usually this is how certificates are stored: in the same file with the private key (since they are useless without it).

```
>>> from cryptography.hazmat.primitives import serialization
>>> private_bytes = private_key.private_bytes(
... encoding=serialization.Encoding.PEM,
... format=serialization.PrivateFormat.TraditionalOpenSSL,
... encryption_algorithm=serialization.NoEncrption())
>>> public_bytes = certificate.public_bytes(
... encoding=serialization.Encoding.PEM)
>>> with open("ca.pem", "wb") as fout:
```

```
...     fout.write(private_bytes + public_bytes)
>>> with open("ca.crt", "wb") as fout:
...     fout.write(public_bytes)
```

This gives us the capability to now be a CA.

In general, for real certificate authorities, we need to generate a Certificate Signing Request (CSR) in order to prove that the owner of the private key actually wants that certificate. However, since we are the certificate authority, we can just create the certificate directly.

There is no difference between creating a private key for a certificate authority and a private key for a service.

```
>>> service_private_key = rsa.generate_private_key(
...     public_exponent=65537,
...     key_size=2048,
...     backend=default_backend()
... )
```

Since we need to sign the *public key*, we need to again calculate it from the private key:

```
>>> service_public_key = service_private_key.public_key()
```

We create a new builder for the service certificate:

```
>>> builder = x509.CertificateBuilder()
```

For services, the `COMMON_NAME` is important; this is what the clients will verify the domain name against.

```
>>> builder = builder.subject_name(x509.Name([
... x509.NameAttribute(NameOID.COMMON_NAME, 'service.test.local')
... ]))
```

We assume that the service will be accessed as `service.test.local`, using some local test resolution. Once again, we limit our certificate validity to about a month.

```
>>> builder = builder.not_valid_before(yesterday)
>>> builder = builder.not_valid_after(next_month)
```

This time, we sign the *service* public key:

```
>>> builder = builder.public_key(public_key)
```

However, we sign *with* the private key of the CA; we do not want *this* certificate to be self-signed.

```
>>> certificate = builder.sign(
...     private_key=private_key, algorithm=hashes.SHA256(),
...     backend=default_backend()
... )
```

Again, we write a PEM file with the key and the certificate:

```
>>> private_bytes = service_private_key.private_bytes(
... encoding=serialization.Encoding.PEM,
... format=serialization.PrivateFormat.TraditionalOpenSSL,
... encryption_algorithm=serialization.NoEncrption())
>>> public_bytes = certificate.public_bytes(
... encoding=serialization.Encoding.PEM)
>>> with open("service.pem", "wb") as fout:
...     fout.write(private_bytes + public_bytes)
```

The `service.pem` file is in a format that can be used by most popular web servers: Apache, Nginx, HAProxy, and many more. It can also be used directly by the Twisted web server, by using the `txsni` extension.

If we add the `ca.crt` file to the trust root, and run, say, an Nginx server, on an IP that our client would resolve from `service.test.local`, then when we connect clients to `https://service.test.local`, they will verify that the certificate is indeed valid.

8.5 Summary

Cryptography is a powerful tool but one that is easy to misuse. By using well-understood high-level functions, we reduce many of the risks in using cryptography. While this does not substitute proper risk analysis and modeling, it does make this exercise somewhat easier.

Python has several third-party libraries with well-vetted code, and it is a good idea to use them.

CHAPTER 9

Paramiko

Paramiko is a library that implements the SSH protocol, commonly used to remotely manage UNIX systems. SSH was originally invented as a secure alternative to the "telnet" command, but soon became the de facto remote management tool. Even systems that use custom agents to manage a server fleet, such as `Salt`, will often be bootstrapped with SSH to install the custom agents. When a system is *described* as "agent-less," as, for example, Ansible is, it usually means that it uses SSH as its underlying management protocol.

Paramiko wraps the protocol and allows both high-level and low-level abstractions. In this chapter, we will concentrate mostly on the high-level abstractions.

Before delving into the details, it is worthwhile to note the synergy that Paramiko has with Jupyter. Using a Jupyter notebook, and running Paramiko inside it, allows a powerful auto-documented remote-control console. The ability to have multiple browsers connected to the same notebook means it has a native ability to share troubleshooting sessions for remote servers, without the need for cumbersome screen sharing.

9.1 SSH Security

The "S" in "SSH" stands for "Secure." The reason to use SSH is because we believe it allows us to *securely* control and configure remote hosts. However, security is a subtle topic. Even if the underlying cryptographic primitives, and the way the protocol uses them, are secure, we must use them properly in order to prevent misusage from causing an issue that opens the door for a successful attack.

It is important to understand how SSH thinks about security in order to use it in a secure way. Unfortunately, it was built at a time when "affordance for security" was not considered a high priority. It is easy to use SSH that negates all security benefits gotten from it.

M. Zadka, *DevOps in Python*, https://doi.org/10.1007/978-1-4842-4433-3_9

The SSH protocol establishes *mutual trust* – the client is assured that the server is authentic, and the server is assured that the client is authentic. There are several ways it can establish this trust, but in this explanation, we will cover the public key method. This is the most common one.

The server's public key is identified by a *fingerprint*. This fingerprint confirms the server's identity in one of two ways. One way is by being communicated by a previously-established secure channel, and saved in a file.

For example, when an AWS EC2 server boots up, it prints the fingerprint to its virtual console. The contents of the console can be retrieved using an AWS API call (which is secured using the web's TLS model) and parsed to retrieve the fingerprint.

The other way, sadly more popular, is the TOFU model – "Trust On First Use." This means that in the initial connection, the fingerprint is assumed to be authentic and stored in a secure location locally. On any subsequent attempts, the fingerprint will be checked against the stored fingerprint, and a different fingerprint will be marked as an error.

The fingerprint is a hash of the server's public key. If the fingerprints are the same, it means the public keys are the same. A server can provide proof that it knows the private key that corresponds to a given public key. In other words, a server can say "here is my fingerprint" and prove that it is indeed a server with that fingerprint. Therefore, if the fingerprint is confirmed, we have established cryptographic trust with the server.

On the other side, users can indicate to the server which public keys they trust. Again, this is often done via some out-of-band mechanism: a web API for the system administrator to put in a public key, a shared filesystem, or a boot script that reads information from the network. Regardless of how it is done, a user's directory can contain a file that means "please authorize connections which can prove they have a private key corresponding to *this* public key as coming from me."

When an SSH connection is established, the client will verify the server's identity as above, and then will provide proof that it owns a private key that corresponds to some public key on the server. If both of these steps succeed, the connection is verified in both directions and can be used for running commands and modifying files.

9.2 Client Keys

Client private and public keys are kept in files that are next to each other. Often users will already have an existing key, but if not, this is easily remedied.

Generating the key itself is easily done from `paramiko`. We choose an `ECDSA` key. The `EC` stands for "elliptic curve." Elliptic curve asymmetric cryptography has better resistance for attacks for the same key size than the older prime number-based cryptography. There is also much less progress in "partial solutions" to EC cryptography, so the consensus in the cryptographic community that they are probably more secure against "nonpublic" actors.

```
>>> from paramiko import ecdsakey
>>> k = ecdsakey.ECDSAKey.generate()
```

As always with asymmetric cryptography, calculating the public part of the key from the private part is fast and straightforward.

```
>>> public_key = k.get_base64()
```

Since this is public, we do not have to worry too much about writing it to a file.

```
>>> with open("key.pub", "w") as fp:
...     fp.write(public_key)
```

However, when we write out the private part of the key, we want to make sure that the file permissions are secure. The way we do that is that after opening the file, but before writing any sensitive data to it, we change the mode.

Note that this is not perfectly safe; the file might have the wrong user if written to the wrong directory, and since some file systems sync the data and metadata separately, a crash at exactly the wrong time can lead to the data being in the file, but with a bad file mode attached. This is only the *minimum* we need to do to safely write a file.

```
>>> import os
>>> with open("key.priv", "w") as fp:
...   os.chmod(0o600, "key.priv")
...   k.write_private_key(fp)
```

We choose the mode `0o600`, which is octal `600`. If we write the bits corresponding to this octal code, they are `110000000`, which translates to `rw-------`: read and write permissions for owner, no permissions for nonowner group members, no permissions for anyone else.

Now through some out-of-band mechanism, we need to push the public key to the relevant server.

For example, depending on the cloud service, code such as:

```
set_user_data(machine_id,
f"""
ssh_authorized_keys:
   - ssh-ecdsa {public_key}
""")
```

where `set_user_data` is implemented using the cloud API, will work on any server that uses cloudinit.

Another thing that is sometimes done is using a Docker container as a bastion. This means we expect users to SSH into the container, and from the container into the specific machine that they need to run commands on.

In this case, a simple `COPY` instruction at build time (or a `docker cp` at runtime, as appropriate) will accomplish the goal. Note that it is perfectly fine to publish to a Docker registry an image with public keys in it – indeed, the requirement that this is a safe operation is part of the definition of public keys.

9.3 Host Identity

As mentioned earlier, the most common, first line of defense against a man-in-the-middle attack in SSH is the so-called TOFU principle – "Trust on First Use." For this to work, after connecting to a host, its fingerprint must be saved in a cache.

The location of that cache used to be straightforward – a file in the user's home directory. However, more modern setups of immutable, throwaway, environments, multiple user machines, and other complications make this more complicated.

It is hard to make a recommendation more general than "share with as many trusted sources as possible." However, to enable that guideline, Paramiko does offer some facilities:

- A client can set a `MissingHostKeyPolicy`, which is any instance that supports an interface. This means that we can have logic to document the key, or query an external database for it.
- An abstraction of the most common format on UNIX systems, the `known_hosts` file. This allows Paramiko to share experiences with keys with the regular SSH client – both by reading it and documenting new entries.

9.4 Connecting

While there are lower-level ways of connecting, the recommended high-level interface is `SSHClient`.

Since `SSHClient` instances need to be closed, it is a good idea (if possible) to use `contextlib.closing` as a context manager:

```
with contextlib.context(SSHClient()) as client:
    client.connect(some_host)
    ## Do things with client
```

Doing this at close to the top level allows us to use `client` as argument to functions, while guaranteeing that it will get closed at the end.

Sometimes, before connecting, we want to configure various policies on the client. This is sometimes useful to do in a function that returns a ready-to-connect client.

These are some of the methods that are useful in preparing to connect relate to the manner of verifying authenticity:

- `load_system_host_keys` will load keys from a source that is managed by other systems. This means they will be used for verifying hosts, but they will not be saved if we choose to save keys.
- `load_host_keys` will load keys from a source that is managed by us. This means that if we choose to save the keys, these will be saved along. For example, we might have a directory with consecutive files, `keys.1`, `keys.2`, `...` and load from the latest one. We can save to the newer file on save, and thus have a safe way to recover from problems (just load a previous version).
- `set_missing_host_policy(policy)` expects an object with the method `missing_host_key`. This method will be called with `client, hostname, key`, and its behavior will determine what to do; an exception will stop the connection, while a successful return will allow the connection to go forward. For example, this can put the host's key in a "tentative" file and raise an exception. The user can look at the tentative file, follow a verification procedure, and add the key to a file loaded by the next iteration.

The connect method takes quite a few arguments. All of them except the hostname are optional. The more important ones are the following:

- hostname – the server to connect to.
- port is needed if we are running on a special port, other than 22. This is sometimes done as part of a security protocol; attempting a connection to port 22 automatically denies all further connections from the IP, while the real server runs on 5022 or a port that is only discoverable via API.
- username – while the default is the local user, this is true less and less. Often cloud virtual machine images have a default "system" user.
- pkey – a private key to use for authentication. This is useful if we want some programmatic way to get the private key (for example, retrieving it from a secret manager).
- allow_agent – this is True by default, for good reasons. This is often a good option, since it means the private key itself will never be loaded by our process, and by extension, cannot be divulged by our process: no accidental logging, debug console, or memory dump is vulnerable.
- look_for_keys – set to False, and give no other key options, to force using an agent.

9.5 Running Commands

The "SH" in SSH stands for shell. The original SSH was invented as a telnet substitute, and its main job is still to run commands on remote machines. Note that "remote" is taken metaphorically, not always literally. SSH is sometimes used to control Virtual Machines, and sometimes even containers, which might be running close by.

After a Paramiko client has connected, it can run commands on the remote host. This is done using the client method exec_command. Note that this method takes the command to be executed *as a string*, not a list. This means that extra care must be exercised when interpolating user values into the command, to make sure that it does not give a user complete execution privileges.

The return value is the standard input, output, and error of the command. This means that the responsibility of communicating carefully with the command, to avoid deadlocks, is firmly in the hands of the end user.

The client also has a method `invoke_shell`, which will create a remote shell and allow programmatic access to it. It returns a `Channel` object, connected directly to the shell. Using the `send` method on the channel will send data to the shell – just as if a person was typing at the terminal. Similarly, the `recv` method allows retrieving the output.

Note that this can be tricky to get right, especially around timing. In general, using `exec_command` is much safer. Opening an explicit shell is rarely needed, unless we need to run commands that work correctly in a terminal. For example, running `visudo` remotely would need a real shell-like access.

9.6 Emulating Shells

We have mentioned that the client has a `invoke_shell`, which will create a remote shell and allow programmatic access to it.

While we can use `send` and `recv` methods on the returned `Channel`, it is sometimes easier to use it as a file.

```
>>> channel = client.invoke_shell()
>>> input = channel.makefile("wb")
>>> output = channel.makefile("rb")
```

Now writing to the input of the command can be done with `input.write`, and reading can be done with `output.write`. Note that this is still subtle: timing and buffering effects can still cause nondeterministic issues.

Notice that it is always possible to `channel.close` and create a new shell, without reconnecting. Therefore, it is a good idea to make sure the shell usage is idempotent. In that case, simple timeouts can help recover from situations where the flow is “stuck,” closing and retrying.

9.7 Remote Files

In order to start file management, we call the client's open_sftp method. This returns an SFTPClient object. We will use methods on this object for all of our remote file manipulation.

Internally, this starts a new SSH channel on the same TCP connection. This means that even while transferring files back and forth, the connection can still be used to send commands to the remote host. SSH does not have a notion of "current directory." Though SFTPClient emulates it, it is better to avoid relying on it and instead use fully qualified paths for all file manipulation. This will make code easier to refactor, and it will not have subtle dependencies on order of operations.

9.7.1 Metadata Management

Sometimes we do not want to change the data, but merely filesystem attributes. The SFTPClient object allows us to do the normal manipulation that we expect.

The chmod method corresponds to os.chmod – it takes the same arguments. Since the second argument to chmod is an integer that is interpreted as a permission bitfield, it is best expressed in octal notation. Thus, the best way to set a file to the "regular" permissions (read/write by owner, read to world) is by this:

```
client.chmod("/etc/some_config", 0o644)
```

Note that the 0644 notation, borrowed from C, does not work in Python 3 (and is deprecated in Python 2). The 0o644 notation is more explicit and Pythonic.

Sadly, nothing protects us from passing in nonsense like this:

```
client.chmod("/etc/some_config", 644)
```

(This would correspond to -w----r-- in a directory listing, which is not insecure – but very confusing!)

Some more metadata manipulation methods are these:

- chown
- listdir_iter – used to retrieve file names and metadata
- stat, lstat – used to retrieve file metadata

- `posix_rename` – used to atomically change a file's name (do not use `rename` – it has confusingly different semantics, and at this point is there for backward compatibility)
- `mkdir`, `rmdir` – create and remove directories
- `utime` – set access and modified times of file

9.7.2 Upload

There are two main ways to upload files to a remote host with Paramiko. One is to simply use `put`. This is definitely the easiest way – give it a local path and a remote path, and it will copy the file. The function also accepts other parameters – mainly a callback to call with intermediate progress. However, in practice, it is better to upload in a different way if such sophistication is required.

The `open` method on an `SFTPClient` returns an open file-like object. It is fairly straightforward to write a loop that copies block by block, or line by line, remotely. In that case, logic for progress could be embedded in the loop itself, instead of having to supply a callback function, and carefully maintain states between calls.

9.7.3 Download

Much like uploading, there are two ways to retrieve files from the remote host. One is via the `get` method, which gets the names of the remote and local files, and manages the copying.

The other is again by using the `open` method, this time in read mode instead of write, and copying block by block or line by line. Again, if a progress indicator is needed, or feedback from the user is desired, that is the better approach.

9.8 Summary

Most UNIX-based servers can be managed remotely using the SSH protocol. Paramiko is a powerful way to automate management tasks in Python, while assuming the least about any server: just that it runs an SSH server, and that we have permissions to log in.

CHAPTER 10

Salt Stack

Salt belongs to a class of systems called "configuration management systems," intended to make administrating a large number of machines easier. It does so by applying the same rules to different machines, making sure that any differences in their configuration are intentional.

It is both written in Python and, more importantly, extensible in Python. For example, wherever a `YAML` file is used, `salt` will allow a Python file that defines a dictionary.

10.1 Salt Basics

The `salt` (or sometimes "SaltStack") system is a *system configuration management* framework. It is designed to bring operating systems into a specific configuration. It is based on the "convergence loop" concept. When running `salt`, it does three things:

- Calculates the desired configuration,
- Calculates how the system differs from the desired configuration,
- Issues commands to bring the system to the desired configuration.

Some extensions to Salt go beyond the "operating system" concept to configure some SaaS products into a desired configuration: for example, there is support for Amazon Web Services, PagerDuty, or some DNS services (those supported by *libcloud*).

Since in a typical environment not all operating systems will need to be configured exactly the same way, Salt allows detecting properties of systems and specifying which configurations apply to which systems. At runtime, Salt uses those to decide what is the complete desired state and enforce it.

M. Zadka, *DevOps in Python*, https://doi.org/10.1007/978-1-4842-4433-3_10

There are a few ways to use `salt`:

- Locally: run a local command that will take the desired steps.
- **SSH: The server will `ssh` into clients and run commands** that will take the desired steps.
- Native protocol: clients will connect to the server and take whatever steps the server instructs them to do.

Using the `ssh` mode removes the need of installing a dedicated client on the remote hosts, since in most configurations, an SSH server is already installed. However, Salt's native protocol for managing remote hosts has several advantages.

For one, it allows the clients to connect to the server, thus simplifying discovery – all we need for discovery is just for clients to the server. It also scales better. Finally, it allows us to control which Python modules are installed in the remote client, which is sometimes essential for Salt extensions.

In the case where some Salt configuration requires an extension that needs a custom module, we can take a hybrid approach: use the SSH-based configuration to bring a host to the point where it knows where the server is, and how to connect to it; and then specify how to bring that host to the desired configuration.

This means there will be two parts to the server: one that uses SSH to bring up the system to a basic configuration that, among other things, has a `salt` client; with the second part waiting for the client to connect in order to send it to the rest of the configuration.

This has the advantage of solving the "secret bootstrapping" problem. We verify the client host's SSH key using a different mechanism, and when connecting to it via Salt, inject the Salt secret to allow the host to connect to it.

When we do choose the hybrid approach, there needs to be a way to find all machines. When using some cloud infrastructure, it is possible to do this using API queries. However we *get* this information, we need to make it accessible to Salt.

This is done using a *roster*. The roster is a YAML file. The top level is the "logical machine name." This is important, since this will be how the machine is addressed using Salt.

```
file_server:              # logical name of machine
    user: moshe           # username
    sudo: True            # boolean
    priv: /usr/local/key  # path to private key
```

```
print_server:             # logical name of machine
    user: moshe           # username
    sudo: True            # boolean
    priv: /usr/local/key2 # path to private key
```

In ideal circumstances, all parameters will be identical for the machines. The `user` is the user to SSH as. The `sudo` boolean is whether sudo is needed: this is almost always True. The only exception is if we SSH as an administrative user (usually root). Since it is a best practice to avoid SSH as root, this is set to `True` in most environments.

The `priv` field is a path to the private key. Alternatively, it can be `agent-forwarding` to use SSH agent. This is often a good idea, since it presents an extra barrier to key leakage.

The roster can go anywhere, but by default Salt will look for it in `/etc/salt/roster`. Putting this file in a different location is subtle: `salt-ssh` will find its configuration, by default, from `/etc/salt/master`. Since the usual reason to put the roster elsewhere is to avoid touching the `/etc/salt` directory, that means we usually need to configure an explicit master configuration file using the `-c` option.

Alternatively, a `Saltfile` can be used. `salt-ssh` will look to a `Saltfile`, in the current directory, for options.

```
salt-ssh:
  config_dir: some/directory
```

If we put in the value `.` in `config_dir`, it will look in the current directory for a `master` file. We can set the `roster_file` field in the `master` file to a local path (for example, `roster`) to make sure the entire configuration is local and locally accessible. This can help if things are being managed by a version control system.

After defining the roster, it is useful to start checking that the Salt system is functioning.

The command

```
$ salt '*' test.ping
```

will send all the machines on the roster (or, later on when we use minions, all connected minions) a ping command. They are all supposed to return True. This command will fail if machines are unreachable, if SSH credentials are wrong, or there are other common configuration problems.

Because this command does not have any effect on the remote machines, it is a good idea to run it first before starting to perform any changes. This will make sure that the system is correctly configured.

There are several other `test` functions, used for more sophisticated checks of the system.

The `test.false` command will intentionally fail. This is useful to see what failures look like. For example, when running Salt via a higher-level abstraction, such as a continuous deployment system, this can be useful to see failures are visible (for example, send appropriate notifications).

The `test.collatz` and `test.fib` functions perform heavy computations and return the time it took as well as the result. This is used to test performance. For example, this might be useful if machines dynamically tune CPU speed according to available power or external temperature, and we want to test whether this is the cause of performance problems.

On the `salt` command line, many things are parsed into Python objects. The interaction of the shell-parsing rules and the `salt`-parsing rules can, at times, be hard to predict. The `test.kwarg` command can be useful when checking how things are parsed. It returns the value that the dictionary passed in as keyword arguments. For example,

```
$ salt '*' test.kwarg word="hello" number=5
                      simple_dict='{thing: 1, other_thing: 2}'
```

will return the dictionary

```
{'word': 'hello', 'number': 5,
 'simple_dict': {'thing': 1, 'other_thing': 2}}
```

Since the combination of the shell-parsing rules and the Salt-parsing rules can be, at times, hard to predict, this is a useful command to be able to debug those combinations and figure out what things are over- or under-quoted.

Instead of `'*'` we can target a specific machine by logical name. This is often useful when seeing a problem with a specific machine; it allows a quick feedback mechanism when trying various fixes (for example, changing firewall settings or SSH private keys).

While testing that the connection works well is important, the reason to use Salt is to actually control machines remotely. While the main usage of Salt is to synchronize to a known state, Salt can also be used to run ad hoc commands.

```
$ salt '*' cmd.run 'mkdir /src'
```

This will cause all connected machines to create a directory `/src`. More sophisticated commands are possible, and again it is possible to only target specific machines.

The technical term for the desired state in Salt is "highstate." The name is a frequent cause of confusion, because it seems to be the opposite of a "low state," which is described almost nowhere. The name "highstate," however, stands for "high-*level* state": it describes the goal of state.

The "low" states, the low-*level* states, are the steps that Salt takes to get to the goal. Since the compilation of the goal to the low-level states is done internally, nothing in the user-facing documentation talks about a "low" state, thus leading to confusion.

The way to apply the desired state is the following:

```
$ salt '*' state.highstate
```

Since there was so much confusion about the name "highstate," in an attempt to reduce the confusion, an alias was created:

```
$ salt '*' state.apply
```

Again, both of these do the *exact same thing*: they figure out what the desired state is, for all machines, and then issue commands to reach it.

The state is described in `sls` files. These files are, usually, in the YAML format and describe the desired state.

The usual way to configure is one file `top.sls` that describes which other files apply to which machines. The `top.sls` name is the name that `salt` will use by default as the top-level file.

A simple homogenous environment might be:

```
# top.sls
base:
  '*':
    - core
    - monitoring
    - kubelet
```

This example would have all machines apply the configuration from `core.sls` (presumably, making sure the basic packages are installed, the right users are configured, etc.); from `monitoring.sls` (presumably, making sure that tools that monitor the machine are installed and running); and `kubelet.sls`, defining how to install and configure the `kubelet`.

Indeed, much of the time Salt will be used to configure machines for workload orchestration tools such as Kubernetes or Docker Swarm.

10.2 Salt Concepts

Salt introduces quite a bit of terminology and also quite a few concepts.

A *minion* is the Salt agent. Even in the "agentless" SSH-based communication, there is still a minion; the first thing that Salt does is send over code for a minion, and then start it.

A Salt *master* sends commands to minions.

A Salt *state* is a file with the `.sls` extension that contains state declarations:

```
name_of_state:
  state.identifier:
    - parameters
    - to
    - state
```

For example:

```
process_tools:
  pkg.installed:
    - pkgs:
      - procps
```

This will make sure the package `procps` (which includes the `ps` command among others) will be installed.

Most Salt states are written to be *idempotent*: to have no effect if they are already in effect. For example, if the package is already installed, Salt will do nothing.

Salt *modules* are different from Python modules. Internally, they do correspond to modules but only some modules.

Unlike *states*, modules *run* things. This means that there is no guarantee, or even attempt at, idempotence.

Often, a Salt state will wrap a module with some logic to decide whether it needs to run the module: for example, before installing a package, `pkg.installed` will check if the package is already installed.

A *pillar* is a way of attaching parameters to specific minions, which can then be reused by different states.

If a *pillar* filters out some minions, then these minions are *guaranteed* to never be exposed to the values in the pillar. This means that pillars are ideal for storing secrets, since they will not be sent to the wrong minions.

For better protection of secrets, it is possible to use gpg to encrypt secrets in pillars. Since gpg is based on asymmetric encryption, it is possible to advertise the public key, for example, in the same source control repository that holds the states and pillars.

This means anyone can add secrets to the configuration, but the private key is needed, on the master, to apply those configurations.

Since GPG is flexible, it is possible to target the encryption to several keys. As a best practice, it is best to load the keys into a `gpg-agent`. This means that when the master needs the secrets, it will use gpg, which will communicate with the `gpg-agent`.

This means the private keys are never exposed to the Salt master directly.

In general, Salt processes directives in states in order. However, a state can always specify `require`. When specifying dependencies, it is best to have the dependent state have a custom, readable, name. This makes dependencies more readable.

```
Extract archive:
  archive.extracted:
    - name: /src/some-files
    - source: /src/some-files.tgz
    - archive_format: tar
  - require:
    - file: Copy archive
Copy archive:
  file.managed:
    - name: /src/some-files.tgz
    - source: salt://some-files.tgz
```

Having explicit readable names helped us make sure we depend on the right state. Note that even though `Extract` precedes `Copy`, it will still wait for the Copy to be finished.

It is also possible to invert the relationship:

```
Extract archive:
  archive.extracted:
    - name: /src/some-files
```

```
    - source: /src/some-files.tgz
    - archive_format: tar
Copy archive:
  file.managed:
    - name: /src/some-files.tgz
    - source: salt://some-files.tgz
  - require_in:
    - archive: Extract archive.
```

In general, much like in this example, inverting the relationship does not improve things. However, this can be sometimes used to minimize or localize changes to files in a shared repository.

There are other relationships possible, and all of them have the ability to be inverted; `onchanges` specifies that the state should only be reapplied if another state has caused actual changes, and `onfail` specifies that the state should only be reapplied if another state application failed. This can be useful to set alerts or make sure that the system goes back to a known state.

There are a few more esoteric relationships possible, like `watch` and `prereq`, which are more specialized.

When using the built-in Salt communication, rather than the SSH method, minions will generate *keys*. Those keys need to be accepted or rejected. One way to do so is to use the `salt-key` command.

As we have mentioned earlier, one way of bootstrapping the trust is to use SSH. In that case, use Salt to transfer over parsed output from running `salt-key -F master` to the minion, and then set it in the minion's configuration under the `master_finger` field.

Similarly, run remotely `salt-call key.finger --local` on the minion (for example, with `salt 'minion' cmd.run`) and compare it to the pending key before accepting. This can be automated, and leads to a verified chain.

There are other ways to bootstrap the trust, depending on what primitives are available. If, for example, hardware key management (HKM) devices are available, they can be used to sign the minions' and the master's keys.

Trusted Platform Modules (TPM) can also be used to mutually assure trust. Both of these mechanisms are beyond our current scope.

Grains (as in, "a grain of salt") parameterize a system. They differ from pillars in that the *minion* decides on the grain; that configuration is stored and modified on the minions.

Some grains, such as `fqdn`, are auto-detected on the minions. It is also possible to define other grains in the minion configuration file.

It is possible to push grains from the master. It is also possible to grab grains from other sources when bootstrapping the minion. For example, on AWS, it is possible to set the `UserData` as a grain.

Salt *environments* are directory hierarchies that each define a separate topfile. Minions can be assigned to an environment, or an environment can be selected when applying the highstate using `salt '*' state.highstate saltenv=...`.

The Salt `file_roots` are a list of directories that function like a path; when looking for a file, Salt will search in them in order, until it finds it. They can be configured on a per-environment basis and are the primary thing distinguishing environments.

10.3 Salt Formats

So far, our example SLS files were YAML files. However, Salt interprets YAML files as *Jinja templates* of YAML files. This is useful when we want to customize fields based on grains or pillars.

For example, the name of the package containing the things we need to build Python packages is different between `CentOS` and `Debian`.

The following SLS snippet shows how to target different packages to different machines in a heterogenous environment.

```
{% if grains['os'] == 'CentOs' %}
python-devel:
{% elif grains['os'] == 'Debian' %}
python-dev:
{% endif %}
  pkg:
    - installed
```

It is important to notice that the Jinja processing step is completely ignorant of the YAML formatting. It treats the file as plain text, does the formatting, and then Salt uses the YAML parser on the result.

This means that it is possible for Jinja to make an invalid file only in some cases. Indeed, we embedded such a bug in the example above; if the OS is neither `CentOS` or `Debian`, the result would be an incorrectly indented YAML file, which will fail to parse in strange ways.

In order to fix it, we would like to raise an explicit exception:

```
{% if grains['os'] == 'CentOs' %}
python-devel:
{% elif grains['os'] == 'Debian' %}
python-dev:
{% else %}
{{ raise('Unrecognized operating system', grains['os']) }}
{% endif %}
  pkg:
    - installed
```

This raises an exception at the right point, in case a machine is added into our roster that is not one of the supported distributions, instead of Salt complaining about a parse error in YAML.

Such care is important whenever doing something nontrivial with Jinja, because the two layers, the Jinja interpolation and the YAML parsing, are not aware of each other. Jinja does not know it is supposed to produce YAML, and the YAML parser does not know what the Jinja source looked like.

Jinja supports *filtering* in order to process values. Some filters are built in to Jinja, but Salt extends it with a custom list.

Among the interesting filters is `YAML_ENCODE`. Sometimes we need to have a *value* in our `.sls` file that is YAML itself: for example, the content of a YAML configuration file that we need copied over.

Embedding YAML in YAML is often unpleasant; special care must be given to proper escaping. With `YAML_ENCODE`, it is possible to encode values written in the native YAML.

For a similar reason, `JSON_ENCODE_DICT` and `JSON_ENCODE_LIST` are useful for systems that take JSON as input.

The list of custom filters is long, and this is one of the frequent things that changes from release to release. The canonical documentation will be on the Salt documentation site, `docs.saltstack.com`, under "Jinja -> Filters."

Though until now we referred to `SLS` files as files that are processed by Jinja and then YAML, this is inaccurate. This is the *default* processing, but it is possible to override the processing with a special instruction.

Salt, itself, only cares that the final result is a YAML-like (or, equivalently in our case, JSON-like) data structure: a dictionary containing recursively dictionaries, lists, strings, and numbers.

The process of converting the text into such a data structure is called "rendering" in Salt parlance. This is opposed to common usage, where rendering means transforming *to* text and parsing means transforming *from* text, so it is important to note when reading Salt documentation.

A thing that can do rendering is a renderer. It is possible to write a custom renderer, but among the built-in renderers, the most interesting one is the `py` renderer.

We indicate that a file should be parsed with the `py` renderer with `#!py` at the top.

In that case, the file is interpreted as a Python file. Salt looks for a function `run`, runs it, and treats the return value as the state.

When running, `__grains__` and `__pillar__` contain the grain and pillar data.

As an example, we can implement the same logic with a `py` renderer.

```
#!py
def run():
    if __grains__['os'] == 'CentOS':
        package_name = 'python-devel'
    elif __grains__['os'] == 'Debian':
        package_name = 'python-dev'
    else:
        raise ValueError("Unrecognized operating system",
                         __grains__['os'])
return { package_name: dict(pkg='installed') }
```

Since the `py` renderer is not a combination of two unrelated parsers, mistakes end up being sometimes easier to diagnose. If we reintroduce the bug from the first version, we get:

```
#!py
def run():
    if __grains__['os'] == 'CentOS':
        package_name = 'python-devel'
    elif __grains__['os'] == 'Debian':
        package_name = 'python-dev'
return { package_name: dict(pkg='installed') }
```

In this case, the result will be a `NameError` pinpointing the erroneous line and the missing name.

The trade-off is that if the configuration is big, and mostly static, reading it in YAML form is more straightforward.

10.4 Salt Extensions

Since Salt is written in Python, it is fully extensible in Python. In general, the easiest way to extend Salt for new kinds of things is to put files in the `file_roots` directory on the Salt master. Unfortunately, there is no package manager for Salt extensions yet. Those files automatically get synchronized to the minions, either when running `state.apply` or when explicitly running `saltutil.sync_state`. The latter is useful if we want to test, for example, a dry run of the state without causing any changes, but *with* the modified modules.

10.4.1 States

State modules go under the root directory for the environment. If we want to share State modules between environments, it is possible to make a custom root and share that root between the right environments.

The following is an example of a module that ensures there are no files that have the name "mean" in them under a specific directory. It is probably not very useful, although in general, making sure that unneeded files are not there could be important. For example, we might want to enforce no `.git` directories.

```
def enforce_no_mean_files(name):
    mean_files = __salt__['files.find'](name, path="*mean*")
    # ...continues below...
```

The name of the function maps to the name of the state in the `SLS` state file. If we put this code in `mean.py`, the appropriate way to address this state would be `mean.enforce_no_mean_files`.

The right way to find files, or indeed, to do anything in a Salt *state* extension, is to call Salt executors. In most non-toy examples, this will mean writing a matching pair: a Salt executor extension and a Salt state extension.

Since we want to progress one thing at a time, we are using a prewritten Salt executor: the `file` module, which has the `find` function.

```
def enforce_no_mean_files(name):
    # ...continued...
    if mean_files = []:
        return dict(
           name=name,
           result=True,
           comment='No mean files detected',
           changes=[],
        )
    # ...continues below...
```

One of the things the state module is responsible for, and indeed often the most important thing, is *doing nothing* if the state is already achieved. This is what being a convergence loop is all about: optimizing for the case of having already achieved convergence.

```
def enforce_no_mean_files(name):
    # ...continued...
    changes = dict(
       old=mean_files,
       new=[],
    )
    # ...continues below...
```

We now know what the changes are going to be. Calculating it here means we can guarantee consistency between the responses in the test vs. non-test mode.

```
def enforce_no_mean_files(name):
    # ...continued...
    changes = dict(
    if __opts__['test']:
        return dict(
           name=name,
           result=None,
```

```
            comment=f"The state of {name} will be changed",
            changes=changes,
        )
    # ...continues below...
```

The next important responsibility is to support the `test` mode. It is considered a best practice to always test before applying a state. We want to clearly articulate the changes that this module will make if activated.

```
def enforce_no_mean_files(name):
    # ...continued...
    changes = dict(
    for fname in mean_files:
        __salt__['file.remove'](fname)
    # ...continues below...
```

In general, we should only be calling one function: the one from the execution module that matches the state module. Since in this example we are using `file` as our execution module, we call the `remove` function in a loop.

```
def enforce_no_mean_files(name):
    # ...continued...
    changes = dict(
    return dict(
        name=name,
        changes=changes,
        result=True,
        comment=f"The state of {name} was changed",
    )
    # ...continues below...
```

Finally, we return a dictionary that has the same changes as the ones documented in the test mode, but with a comment indicating that these have already run.

This is the typical structure of a state module: one (or more) functions, which accept a name (and possibly more arguments) and then return a result. The structures of "check if changes are needed," "check if we are in test mode," and then "actually perform changes" are also typical.

10.4.2 Execution

For historical reasons, execution modules go in the file root's `_modules` subdirectory. Similarly to execution modules, they are also synchronized when `state.highstate` is applied, as well as when explicitly synchronized via `saltutil.sync_all`.

As an example, we write an execution module to delete several files, in order to simplify our state module above.

```
def multiremove(files):
    for fname in files:
        __salt__['file.remove'](fname)
```

Note that `__salt__` is usable in execution modules as well. However, while it can cross-call other execution modules (in this example, `file`) it cannot cross-call into state modules.

We put this code in `_modules/multifile`, and we can change our state module to have:

```
__salt__['multifile.mutiremove'](mean_files)
```

instead of

```
for fname in mean_files:
    __salt__['file.remove'](fname)
```

Execution modules are often simpler than state modules, as in this example. In this toy example, the execution module barely does anything at all, just coordinating calls to other execution modules.

This is not completely atypical, however. Salt has so much logic for managing machines that often all an execution module has to do is just to coordinate calls to other execution modules.

10.4.3 Utility

When writing several execution or state modules, it sometimes is the case that there is some common code that can be factored out.

This code can sit in so-called "utility modules." Utility modules sit under the file root's `_utils` directory and will be available as the `__utils__` dictionary.

As an example, we can factor out the calculation of the return value in the state module:

```
def return_value(name, old_files):
    if len(old_files) == 0:
        comment = "No changes made"
        result = True
    elif __opts__['test']:
        comment = f"{name} will be changed"
        result = None
    else:
        comment = f"{name} has been changed"
        result = True
    changes = dict(old=old_files, new=[])
    return dict(
        name=name,
        comment=comment,
        result=result,
        changes=changes,
    )
```

If we use the execution module and the utility modules, we get a simpler state module:

```
def enforce_no_mean_files(name):
    mean_files = __salt__['files.find'](name, path="*mean*")
    if len(mean_files) == 0 or __opts__['test']:
        return __utils__['removal.return_value'](name, mean_files)
    __salt__['multifile.mutiremove'](mean_files)
    return __utils__['removal.return_value'](name, mean_files)
```

In this case, we could have put the function as a regular function in the module; putting it in a utility module was used to show how to call functions in utility modules.

10.4.4 Extra Third-Party Dependencies

Sometimes it is useful to have third-party dependencies, especially when writing new state and execution modules. This is straightforward to do when installing a minion; we just make sure to install the minion in a virtual environment with those third-party dependencies.

When using Salt with SSH, this is significantly less trivial. In that case, it is sometimes best to bootstrap from SSH to a real minion. One way to achieve that is to have a persistent state in the SSH "minion" directory, and have the installation of the minion set a grain of "completely_disable" in the SSH minion. This would make sure that the SSH configuration does not cross-talk with the regular minion configuration.

10.5 Summary

Salt is a Python-based configuration management system. For nontrivial configurations, it is possible to *express* the desired system configuration using Python, which can sometimes be more efficient than templating YAML files. It is also possible to *extend* it with Python in order to define new primitives.

CHAPTER 11

Ansible

Like Salt, Ansible is another configuration management system. However, Ansible does not have a custom agent: it always works with SSH. Unlike the way Salt works with SSH, where it spins up an ad hoc minion and sends it commands, Ansible calculates the commands on the server and sends simple commands and files through the SSH connection.

By default, Ansible will try to use the local SSH command as the control machine. If the local command is unsuitable for any reason, Ansible will fall back to using the Paramiko library.

11.1 Ansible Basics

Ansible can be installed using `pip install ansible` in a virtual environment. After installing it, the simplest thing is to ping the localhost:

```
$ ansible localhost -m ping
```

This is useful, since if this works it means quite a few things are configured correctly: running the SSH command, configuring the SSH keys, and the SSH host keys.

In general, the best way to use Ansible, as always when using SSH communication, is with a locally-encrypted private key that is loaded into an SSH agent. Since `ansible`, by default, will use the local SSH command, if `ssh localhost` works the right way (without asking for a password), then Ansible will work correctly. If *localhost* is not running an SSH daemon, replace the examples below with a separate Linux host: possibly one running as a virtual machine locally.

Slightly more sophisticated, but still not requiring a complicated setup, is running a specific command:

```
$ ansible localhost -a "/bin/echo hello world"
```


M. Zadka, *DevOps in Python*, https://doi.org/10.1007/978-1-4842-4433-3_11

We can also give an explicit address:

```
$ ansible 10.40.32.195 -m ping
```

will try to SSH to 10.40.42.195.

The set of hosts Ansible will try to access by default is called the "inventory." The inventory can either be specified statically in an INI or YAML files. However, the more common option is to write an "inventory script," which generates the list of machines.

An inventory script is simply a Python file that can be run with the arguments --list and -host <hostname>. Ansible will use, by default, the same Python used to run it in order to run the inventory script. It is possible to make the inventory script a "real script" running with any interpreter, like a different version of Python, by adding a shebang line. Traditionally, the file is not named with .py. Among other things, this avoids accidental imports of the file.

When run with --list, it is supposed to output the inventory as formatted JSON. When run with --host, it is supposed to print the variables for the host. In general, it is perfectly acceptable to always print an empty dictionary in these circumstances.

Here is a simple inventory script:

```
import sys

if '--host' in sys.argv[1:]:
    print(json.dumps({}))

print(json.dumps(dict(all='localhost')))
```

This inventory script is not very dynamic; it will always print the same thing. However, it is a valid inventory script.

We can use it with

```
$ ansible -i simple.inv all -m ping
```

This will again ping (using SSH) the localhost.

Ansible is not primarily used to run ad hoc commands against hosts. It is designed to run "playbooks." Playbooks are YAML files that describe "tasks."

```
---
- hosts: all
  tasks:
    - name: hello printer
      shell: echo "hello world"
```

This playbook will run `echo "hello world"` on all connected hosts.

In order to run it, with the inventory script we created,

```
$ ansible-playbook -i simple.inv echo.yml
```

In general, this will be the most common command to use when running Ansible day to day. Other commands are mostly used as debugging and troubleshooting, but in normal circumstances, the flow is to rerun the playbook "a lot."

By "a lot," we mean that, in general, playbooks should be written to be safely idempotent; executing the same playbook in the same circumstances again should not have any effect. Note that in Ansible, idempotency is a property of the *playbook*, not of the basic building blocks.

For example, the following playbook is not idempotent:

```
---
- hosts: all
  tasks:
    - name: hello printer
      shell: echo "hello world" >> /etc/hello
```

One way to make it idempotent is to make it notice the file is already there:

```
---
- hosts: all
  tasks:
    - name: hello printer
      shell: echo "hello world" >> /etc/hello
      creates: /etc/hello
```

This will notice the file exists and will skip the command if so.

In general, in more complex settings, instead of listing tasks in the playbooks, these will be delegated to *roles*.

Roles are a way of separating concerns and flexibly combining them per host.

```
---
- hosts: all
  roles:
    - common
```

Then, under roles/common/tasks/main.yml

```
---
- hosts: all
  tasks:
    - name: hello printer
      shell: echo "hello world" >> /etc/hello
      creates: /etc/hello
```

This will do the same thing as above, but now it is indirected through more files. The benefit is that if we have many different hosts, and we need to combine instructions for some of them, this is a convenient platform to define parts of more complicated setups.

11.2 Ansible Concepts

When Ansible needs to use secrets, it has its internal "vault." The vault has encrypted secrets and is decrypted with a password. Sometimes this password will be in a file (ideally on an encrypted volume).

Ansible roles and playbooks are *jinja2* YAML files. This means they can use interpolation, and they support a number of Jinja2 filters.

Some useful ones are `from/to_json/yaml`, which allow data to be parsed and serialized back and forth. The `map` filter is a meta-filter that applies a filter item by item to an iterable object.

Inside the filters, there is a set of variables defined. Variables can come from multiple sources: the Vault (for secrets), directly in the playbook or role, or in files included from it. Variables can also come from the inventory (which can be useful if different inventories are used with the same playbook). The `ansible_facts` variable is a dictionary that has the facts about the current host: operating system, IP, and more.

They can also be defined directly on the command line. While this is dangerous, it can be useful for quick iterations.

In playbooks, it is often the case that we will need to define both which user to *log in* as, as well as which user (usually root) to execute tasks as.

All of those can be configured on a playbook and overridden on a per-task level.

The user that we *log in* as is `remote_user`. The user that we execute as is either `remote_user` if `become` is `False`, or `become_user` if `become` is `True`. If `become` is `True`, the user switching will be done by `become_method`.

The defaults are:

- `remote_user` – same as local user
- `become_user - root`
- `become - False`
- `become_method - sudo`

These defaults are usually correct, except for `become`, which often needs to be overridden to `True`. In general, it is best to configure machines so that, whatever we chose the `become_method` to be, the process of user switching does not require passwords.

For example, the following will work on common cloud-providers' versions of Ubuntu:

```
- hosts: databases
  remote_user: ubuntu
  become: True

  tasks:
  - name: ensure that postgresql is started
    service:
      name: postgresql
      state: started
```

If this is impossible, we need to give the argument `--ask-become-pass` to have Ansible ask for the credentials at runtime. Note that while this works, this will hamper automation attempts, and it is best to avoid it.

Ansible supports "patterns" to indicate which hosts to update. In `ansible-playbook`, this is done with `--limit`. It is possible to do set arithmetic on groups: `:` means "union," `:!` means "set difference," and `:&` means "intersection." In that case, the basic sets are the sets as defined in the inventory. For example, `databases:!mysql` will limit the command to only `databases` hosts that are not `mysql`.

Patterns can be regular expressions that match hosts' names or IPs.

11.3 Ansible Extensions

We have seen one way to extend ansible using custom Python code: dynamic inventory. In the dynamic inventory example, we wrote an ad hoc script. The script, however, was run as a separate process. A better way to extend Ansible, and one that generalizes beyond inventory, is to use *plugins*.

An *inventory plugin* is a Python file. There are several places for this file so that Ansible can find it: often the easiest is `plugins/inventory_plugins` in the same directory as the playbook and roles.

This file should define a class called `InventoryModule` that inherits from `BaseInventoryPlugin`. The class should define two methods: `verify_file` and `parse`. The `verify_file` function is mostly an optimization; it is meant to quickly skip the parsing if the file is not the right one for the plugin. It is an optimization since `parse` can (and should) raise `AnsibleParserError` if the file cannot be parsed for any reason. Ansible will then try the other inventory plugins.

The `parse` function signature is

```
def parse(self, inventory, loader, path, cache=True):
    pass
```

A simple example parsing JSON:

```
def parse(self, inventory, loader, path, cache=True):
    super(InventoryModule, self).parse(inventory, loader, path, cache)
    try:
        with open(path) as fpin:
            data = json.loads(fpin.read())
    except ValueError as exc:
        raise AnsibleParseError(exc)
    for host in data:
        self.inventory.add_host(server['name'])
```

The `inventory` object is how to manage the inventory; it has methods for `add_group`; `add_child`; and `set_variable`, which is how the inventory is extended.

The `loader` is a flexible loader that can guess a file's format and load it. The `path` is the path to the file that has the plugin parameters. Notice that in some cases, if the plugin is specific enough, the parameters and the loader might not be needed.

The other common plugin to write is a “lookup” plugin. Lookup plugins can be called from the Jinja2 templates in Ansible, in order to do arbitrary computation. This is often a good alternative when templates start getting a little too complicated. Jinja2 does not scale well to complex algorithm, or to call into third-party libraries easily.

Lookup plugins are sometimes used for complex computation, and sometimes for calling into a library to allow computing a parameter in a role. For example, it can take the name of an environment and calculate (based on local conventions) what are the related objects.

```
class LookupModule(LookupBase):

    def run(self, terms, variables=None, **kwargs):
        pass
```

For example, we can write a lookup plugin that calculates the largest common path of several paths:

```
class LookupModule(LookupBase):

    def run(self, terms, variables=None, **kwargs):
        return os.path.commonpath(terms)
```

Note that when *using* lookup modules, both `lookup` and `query` can be used from Jinja2. By default, `lookup` will convert the return value into a string. The parameter `wantslist` can be sent to avoid a conversion if the return value is a list. Even in that case, it is important to only return a “simple” object: something composed only of integers, floats and strings, or lists and dictionaries thereof. Custom classes will be coerced into strings, in various surprising ways.

11.4 Summary

Ansible is a simple configuration management that is easy to set up, requiring just SSH access. Writing new inventory plugins and lookup plugins allows implementing custom processing with little overhead.

CHAPTER 12

Docker

Docker is a system for application-level virtualization. While different Docker *containers* share a *kernel*, they will usually share little else: files, processes, and more can all be separate. It is commonly used for both testing software systems and running them in production.

There are two main ways to automate Docker. It is possible to use the `subprocess` library and use the `docker` command line. This is a popular way and does have some advantages.

However, an alternative is to use the `dockerpy` library. This allows doing some things that are completely impossible with the `docker` command, as well as some things that are merely impossible or annoying with the command.

One of the advantages is installation; getting DockerPy installed is just `pip install docker` in a virtual environment, or any of the other ways to install Python packages. Installing the Docker binary client is often more involved. While it does come when installing the Docker daemon, it is not uncommon to need just a client, while the server runs on a different host.

When using the `docker` Python package, the usual way is to connect using

```
import docker
client = docker.from_env()
```

This will, by default, connect to the local Docker daemon. However, in an environment that has been prepared using `docker-machine env`, for example, it will connect to the relevant remote Docker daemon.

In general, `from_env` will use an algorithm that is compatible with the one the `docker` command-line client uses, and so is often useful in a drop-in replacement.

For example, this is useful in Continuous Integration environments that allocate a Docker host per CI session. Since they will set up the local environment to be compatible with the `docker` command, `from_env` will do the right thing.

M. Zadka, *DevOps in Python*, https://doi.org/10.1007/978-1-4842-4433-3_12

It is also possible to connect directly using the host's details. The `DockerClient` constructor will do that.

12.1 Image Building

The client's `images` attribute's `build` method accepts a few arguments that allow doing things that are hard from the command line. The method accepts only keyword arguments. There are *no* required arguments, also at least one of `path` and `fileobj` must be passed in.

The `fileobj` parameter can point to a file-like object that is a tarball (or a gzipped tarball, in which case the `encoding` parameter needs to be set to `gzip`). This will end up being the build context, and the `dockerfile` parameter will be taken to mean a path inside the context. This allows creating the build context explicitly.

With the usual `docker build` command, the context is the contents of a directory, further going through an include/exclude with a `.dockerignore` file. Generating the tarball in memory using `BytesIO` and the `tarfile` library means the contents can be explicit. Note that this means that Python can also generate the `Dockerfile` in memory.

This way, there is no need to create *external* files; the entire build system is specified in Python and passed directly to Docker.

For example, here is a simple program to create a Docker image that has nothing except a file `/hello` with a simple greeting:

```
fpout = io.BytesIO()
tfout = tarfile.open(fpout, "w|")
info = tarfile.TarInfo(name="Dockerfile")
dockerfile = io.Bytes("FROM scratch\nCOPY hello /hello".encode("ascii"))
tfout.addfile(tarinfo=info, fileobj=dockerfile)
hello = io.Bytes("This is a file saying 'hello'".encode("ascii"))
info = tarfile.TarInfo(name="hello")
tfout.addfile(tarinfo=info, fileobj=hello)
fpout.seek(0)
client.build(fileobj=fpout, tag="hello")
```

Note that this image is naturally not very useful. We cannot create a running container from it, since it has no executables. However, this simple example does show we can create a Docker image without having any external files.

This can come in handy, for example, when creating an image from wheels; we can download the wheels into in-memory buffers, create the container, tag it, and push it, all without needing any temporary files.

12.2 Running

The `containers` attribute on the client allows managing running containers.

The `containers.run()` method will run a container. The arguments are much the same as the `docker run` command line, but there are some differences in the best way to use them.

It is almost always a good idea, from Python, to use the `detach=True` option. This will cause `run()` to return a `Container` object. If you need to wait until it exits, for some reason, call `.wait`, explicitly, on the container object.

This allows timeouts, which are useful for killing runaway processes. The return value of the container object also allows retrieving the logs, or inspecting the list of processes inside the container.

The `containers.create` method will create a container, but not run it, like `docker create`.

Regardless of whether a container is running or not, it is possible to interact with its file system. The `get_archive` method will retrieve a file or recursively a directory from the container. It will return a tuple. The first element is an iterator that yields raw bytes objects. Those can be parsed as a tar archive. The second element is a dictionary containing metadata about the file or the directory.

The `put_archive` command injects files into the container. This is sometimes useful between `create` and `start` to fine-tune the container: for example, injecting a configuration file for a server.

It is even possible to use this as an alternative to the build command; a combination of `container.put_archive` and `container.commit` with `containers.run` and `containers.create` allows building containers incrementally, without a Dockerfile. One advantage of this approach is that the layer division is orthogonal to the number of steps: you can have several logical steps be the same layer.

Note, however, that deciding which "layers" to cache becomes our responsibility, in this case. Also, in this case, the "intermediate layers" are fully fledged images. This has its advantages: for example, cleaning up becomes more straightforward.

12.3 Image Management

The client's `images` attribute allows manipulating the container images. The `list` method on the attribute returns a list of images. These are image *objects*, not just names. An image can be retagged with the method `tag`. This allows, for example, tagging a specific image as the `:latest`.

The `pull()` and `push()` methods correspond to the docker client pull and push. The `remove()` command allows removing images. Note that the argument is a name, not an `Image` object.

For example, here is a simple example that will retag the latest image as `latest`:

```
images = client.list(name="myusername/myrepo")
sofar = None
for image in images:
    maxtag = max(tag for tag in image.tags if tag.startswith("date-"))
    if sofar is None or maxtag > sofar:
        sofar = maxtag
        latest = image
latest.tag("myusername/myrepo", tag="latest")
client.push("myusername/myrepo", tag="latest")
```

12.4 Summary

Using the `dockerpy` is a powerful alternative for the Docker client when automating Docker. It allows us to use the full power of Python, including string manipulation and buffer manipulation, to build a container image, run a container, and manage running containers.

CHAPTER 13

Amazon Web Services

Amazon Web Services, AWS, is a cloud platform. It allows using computation and storage resources in a data center, paying by usage. One of the central principles of AWS is that all interactions with it should be possible via an API: the web console, where computation resources can be manipulated, is just another front end to the API. This allows automating configuration of the infrastructure: so-called "infrastructure as code," where the computing infrastructure is reserved and manipulated programmatically.

The Amazon Web Services team supports a package on PyPI, `boto3`, to automate AWS operations. In general, this is one of the best ways to interact with AWS.

While AWS does support a console UI, it is usually best to use that as a read-only window into AWS services. When making changes through the console UI, there is no repeatable record of it. While it is possible to log actions, this does not help to reproduce them.

Combining `boto3` with Jupyter, as we have discussed in an early chapter, makes for a powerful AWS operations console. Actions taken through Jupyter, using the `boto3` API, can be repeated, automated, and parameterized as needed.

When making ad hoc changes to the AWS setup to solve a problem, it is possible to attach the notebook to the ticket tracking the problems, so that there is a clear record of what was done to address the problem. This serves both to understand what was done in case this caused some unforeseen issues, and to easily repeat this intervention in case this solution is needed again.

As always, notebooks are not an *auditing* solution; for one, when allowing access via `boto3`, actions do not have to be performed via a notebook. AWS has internal ways to generate *audit* logs. The notebooks are there to document intent and allow repeatability.

M. Zadka, *DevOps in Python*, https://doi.org/10.1007/978-1-4842-4433-3_13

13.1 Security

For automated operations, AWS requires *access keys*. Access keys can be configured for the root account, but this is not a good idea. There are no restrictions possible on the root account, so this means that these access keys can do everything.

The AWS platform for roles and permissions is called “Identity and Access Management,” or IAM. The IAM service is responsible for users, roles, and policies.

In general, it is better to have a separate IAM user for each human user, as well as for each automated task that needs to be taken. Even if they all share an access policy, having distinct users means it is easier to do key management, as well as having accurate audit logs of who (or what) did what.

13.1.1 Configuring Access Keys

With the right security policy, users can be in charge of their own access keys. A single “access key” is composed of the access key *ID* and the access key *secret*. The *ID* does not need to be kept secret, and it will remain accessible via IAM user interface after generation. This allows, for example, disabling or deleting an access key by ID.

A user can configure up to two access keys. Having two keys allows doing 0-downtime key rotations. The first step is to generate a new key. Then replace the old key everywhere. Afterwards, disable the old key. *Disabling* the old key will make anything that tries to use it fail. If such a failure is detected, it is easy to *re-enable* the old key, until the task using that key can be upgraded to the new key.

After a certain amount of time, when no failures have been observed, it should be safe to delete the old key.

In general, local security policies determine how often keys should be rotated, but this should usually be at least a yearly ritual. In general, this should follow practices for other API secrets used in the organization.

Note that in AWS, different computation tasks can have their own IAM credentials.

For example, an EC2 machine can be assigned an IAM role. Other higher-level computation tasks can also be assigned a role. For example, an Elastic Container Service (ECS) task, which runs one or more Docker containers, can be assigned an IAM role. So-called “serverless” Lambda functions, which run on infrastructure allocated on an as-needed basis, can also be assigned an IAM role.

The `boto3` client will automatically use these credentials if running from such a task. This removes the need to explicitly manage credentials, and it is often a safer alternative.

13.1.2 Creating Short-Term Tokens

AWS supports something called "Short-Term Tokens" or STS. Short-term tokens can be used for several things. They can be used to convert alternative authentication methods into tokens that can be used with any `boto3`-based program, for example, by putting them in an environment variable.

For example, in an account that has been configured with SSO-based authentication based on SAML, `boto3.client('sts').assume_role_with_saml` can be called to generate a short-term security token. This can be used in `boto3.Session` in order to get a session that has those permissions.

```
import boto3
response = boto3.client('sts').assume_role_with_saml(
    RoleArn=role_arn,
    PrincipalArn=principle_arn,
    SAMLAssertion=saml_assertion,
    DurationSeconds=120
)
credentials = response['Credentials']
session = boto3.Session(
    aws_access_key_id=credentials['AccessKeyId'],
    aws_secret_access_key=credentials['SecretAccessKey'],
    aws_session_token=credentials['SessionToken'],
)
print(session.client('ec2').describe_instances())
```

A more realistic use case would be in a custom web portal that is authenticated to an SSO portal. It can perform actions on behalf of the user, without *itself* having any special access privileges to AWS.

On an account that has been configured with cross-account access, `assume_token` can return credentials for the granting account.

Even when using a single account, sometimes it is useful to create a short-term token. For example, this can be used to limit permissions; it is possible to create an STS with a limited security policy. Using those limiting tokens in a piece of code that is more prone to vulnerabilities, for example, because of direct user interactions, allows limiting the attack surface.

13.2 Elastic Computing Cloud (EC2)

The Elastic Computing Cloud (EC2) is the most basic way to access compute (CPU and memory) resources in AWS. EC2 runs "machines" of various types. Most of those are "virtual machines" (VMs) that run, together with other VMs, on physical hosts. The AWS infrastructure takes care of dividing resources between the VMs in a fair way.

The EC2 service also handles the resources that machines need to work properly: operating system images, attached storage, and networking configuration, among others.

13.2.1 Regions

EC2 machines run in "regions." Regions usually have a human-friendly name (such as "Oregon") and an identifier that is used for programs (such as "us-west-2").

There are several regions in the United States: at time of writing, North Virginia ("us-east-1"), Ohio ("us-east-2"), North California ("us-west-1"), and Oregon ("us-west-2"). There are also several regions in Europe, Asia Pacific, and more.

When we connect to AWS, we connect to the region we need to manipulate: `boto3.client("ec2", region_name="us-west-2")` returns a client that connects to the Oregon AWS data center.

It is possible to specify default regions in environment variables and configuration files, but it is often the best options to be explicit in code (or retrieve it from higher-level application configuration data).

EC2 machines also run in an *availability zone*. Note that while regions are "objective" (every customer sees the region the same), availability zones are not: one customer's "us-west-2a" might be another's "us-west-2c."

Amazon puts all EC2 machines into some Virtual Private Cloud (VPC) private network. For simple cases, an account will have one VPC per region, and all EC2 machines belonging to that account will be in that VPC.

A *subnet* is how a VPC intersects with an availability zone. All machines in a subnet belong to the same zone. A VPC can have one or more *security groups*. Security groups can have various firewall rules set up about what network connections are allowed.

13.2.2 Amazon Machine Images

In order to start an EC2 machine, we need an "operating system image." While it is possible to build custom Amazon Machine Images (AMIs), it is often the case we can use a ready-made one.

There are AMIs for all major Linux distributions. The AMI ID for the right distribution depends on the AWS *region* in which we want to run the machine. Once we have decided on the region and on the distribution version, we need to find the AMI ID.

The ID can sometimes be nontrivial to find. If you have the *product code*, for example, `aw0evgkw8e5c1q413zgy5pjce`, we can use `describe_images`.

```
client = boto3.client(region_name='us-west-2')
description = client.describe_images(Filters=[{
    'Name': 'product-code',
    'Values': ['aw0evgkw8e5c1q413zgy5pjce']
}])
print(description)
```

The CentOS wiki contains product codes for all relevant CentOS versions.

AMI IDs for Debian images can be found on the Debian wiki. The Ubuntu website has a tool to find the AMI IDs for various Ubuntu images, based on region and version. Unfortunately, there is no centralized automated registry. It is possible to search for AMIs with the UI, but this is risky; the best way to guarantee the authenticity of the AMI is to look at the creator's website.

13.2.3 SSH Keys

For ad hoc administration and troubleshooting, it is useful to be able to SSH into the EC2 machine. This might be for manual SSH, using Paramiko, Ansible, or bootstrapping Salt.

Best practices for building AMIs that are followed by all major distributions for their default images use `cloud-init` to initialize the machine. One of the things `cloud-init` will do is allow a preconfigured user to log in via an SSH public key that is retrieved from the so-called "user data" of the machine.

Public SSH keys are stored by region and account. There are two ways to add an SSH key: letting AWS generate a key pair, and retrieving the private key, or generating a key pair ourselves and pushing the public key to AWS.

The first way is done with the following:

```
key = boto3.client("ec2").create_key_pair(KeyName="high-security")
fname = os.path.expanduser("~/.ssh/high-security")
with open(fname, "w") as fpout:
    os.chmod(fname, 0o600)
    fpout.write(key["KeyMaterial"])
```

Note that they keys are ASCII encoded, so using string (rather than byte) functions is safe.

Note that it is a good idea to change the file's permissions *before* putting in sensitive data. We also store it in a directory that tends to have conservative access permissions.

If we want to import a public key to AWS, we can do it with this:

```
fname = os.path.expanduser("~/.ssh/id_rsa.pub")
with open(fname, "rb") as fpin:
    pubkey = fpin.read()
encoded = base64.encodebytes(pubkey)
key = boto3.client("ec2").import_key_pair(
    KeyName="high-security",
    PublicKeyMaterial=encoded,
)
```

As explained in the cryptography chapter, having the private key on as few machines as possible is best.

In general, this is a better way. If we generate keys locally and encrypt them, there are fewer places where an unencrypted private key can leak from.

13.2.4 Bringing Up Machines

The run_instances method on the EC2 client can start new instances.

```
client = boto3.client("ec2")
client.run_instances(
    ImageId='ami-d2c924b2',
    MinCount=1,
    MaxCount=1,
```

```
    InstanceType='t2.micro',
    KeyName=ssh_key_name,
    SecurityGroupIds=['sg-03eb2567']
)
```

The API is a little counterintuitive – in almost all cases, both `MinCount` and `MaxCount` need to be 1. For running several identical machines, it is much better to use an AutoScaling Group (ASG), which is beyond the scope of the current chapter. In general, it is worth remembering that as AWS's first service, EC2 has the oldest API, with the least lessons learned on designing good cloud automation APIs.

While in general the API allows running more than one instance, this is not often done. The `SecurityGroupIds` imply which VPC the machine is in. When running a machine from the AWS console, a fairly liberal security group is automatically created. For debugging purposes, using this security group is a useful shortcut, although in general it is better to create custom security groups.

The AMI chosen here is a CentOS AMI. While `KeyName` is not mandatory, it is highly recommended to create a key pair, or import one, and use the name.

The `InstanceType` indicates the amounts of computation resources allocated to the instance. `t2.micro` is, as the name implies, is a fairly minimal machine. It is useful mainly for prototyping but usually cannot support all but the most minimal production workloads.

13.2.5 Securely Logging In

When logging in via SSH, it is a good idea to know beforehand what is the public key we expect. Otherwise, an intermediary can hijack the connection. Especially in cloud environments, the "Trust-on-First-Use" approach is problematic; there are a lot of "first uses" whenever we create a new machine. Since VMs are best treated as disposable, the TOFU principle is of little help.

The main technique in retrieving the key is to realize that the key is written to the "console" as the instance boots up. AWS has a way for us to retrieve the console output:

```
client = boto3.client('ec2')
output = client.get_console_output(InstanceId=sys.argv[1])
result = output['Output']
```

Unfortunately, boot-time diagnostic messages are not well structured, so the parsing must be somewhat ad hoc.

```
rsa = next(line
           for line in result.splitlines()
           if line.startswith('ssh-rsa'))
```

We look for the first line that starts with `ssh-rsa`. Now that we have the public key, there are several things we can do with it. If we just want to run an SSH command line, and the machine is not VPN-accessible-only, we will want to store the public IP in `known_hosts`.

This avoids a Trust-on-First-Use (TOFU) situation: `boto3` uses Certificate Authorities to connect securely to AWS, and so the SSH key's integrity is guaranteed. Especially for cloud platforms, TOFU is a poor security model. Since it is so easy to create and destroy machines, the lifetime of machines is sometimes measured in weeks or even days.

```
resource = boto3.resource('ec2')
instance = resource.Instance(sys.argv[1])
known_hosts = (f'{instance.public_dns_name},'
               f'{instance.public_ip_address} {rsa}')
with open(os.path.expanduser('~/.ssh/known_hosts'), 'a') as fp:
    fp.write(known_hosts)
```

13.2.6 Building Images

Building your own images can be useful. One reason to do it is to accelerate startup. Instead of booting up a vanilla Linux distribution and then installing needed packages, setting configuration, and so on, it is possible to do it once, store the AMI, and then launch instances from this AMI.

Another reason to do it is to have known upgrade times; running `apt-get update && apt-get upgrade` means getting the latest packages at time of upgrade. Instead, doing this in an AMI build allows knowing all machines are running from the same AMI. Upgrades can be done by first replacing some machines with machines with the new AMI, checking the status, and then replacing the rest. This technique, used by Netflix among others, is called "immutable images." While there are other approaches to immutability, this is one of the first ones that was successfully deployed in production.

One way to prepare machines is to use a configuration management system. Both Ansible and Salt have a “local” mode that runs commands locally, instead of via a server/client connection.

The steps are:

- Launching an EC2 machine with the right base image (for example, vanilla CentOS).
- Retrieve the host key for securely connecting.
- Copy over Salt code.
- Copy over Salt configuration.
- Via SSH, run Salt on the EC2 machine.
- At the end, call `client("ec2").create_image` in order to save the current disk contents as an AMI.

```
$ pex -o salt-call -c salt-call salt-ssh
$ scp -r salt-call salt-files $USER@$IP:/
$ ssh $USER@$IP /salt-call --local --file-root /salt-files
(botovenv)$ python
...
>>> client.create_image(....)
```

This approach means a simple script, running on a local machine or in a CI environment, can generate an AMI from source code.

13.3 Simple Storage Service (S3)

The simple storage service (S3) is an object storage service. Objects, which are byte streams, can be stored and retrieved. This can be used to store backups, compressed log files, video files, and similar things.

S3 stores objects in *buckets*, by *key* (a string). Objects can be stored, retrieved, or deleted. However, objects cannot be modified in place.

S3 buckets names must be globally unique, not just per account. This uniqueness is often accomplished by adding the account holder’s domain name, for example, `large-videos.production.example.com`.

Buckets can be set to be publicly available, in which case objects can be retrieved by accessing a URL composed of the bucket's name and the object's name. This allows S3 buckets, properly configured, to be static websites.

13.3.1 Managing Buckets

In general, bucket creation is a fairly rare operation. New buckets correspond to new code *flows*, not code *runs*. This is partially because buckets need to have unique names. However, it is sometimes useful to create buckets automatically, perhaps for many parallel test environments.

```
response = client("s3").create_bucket(
    ACL='private',
    Bucket='my.unique.name.example.com',
)
```

There are other options, but those are usually not needed. Some of those have to do with granting permissions on the bucket. In general, a better way to manage bucket permissions is the way all permissions are managed: by attaching policies to roles or IAM users.

In order to list possible keys, we can use this:

```
response = client("s3").list_objects(
    Bucket=bucket,
    MaxKeys=10,
    Marker=marker,
    Prefix=prefix,
)
```

The first two arguments are important; it is necessary to specify the bucket, and it is a good idea to make sure that responses are of known maximum size.

The `Prefix` parameter is useful especially when we use the S3 bucket to simulate a "file system." For example, this is what S3 buckets that are served as websites usually look like. When exporting CloudWatch logs to S3, it is possible to specify a prefix, exactly to simulate a "file system." While internally the bucket is still flat, we can use something like `Prefix="2018/12/04/"` to get only the logs from December 4th, 2018.

When a there are more objects that qualify than MaxKeys, the response will be truncated. In that case, the IsTruncated field in the response will be True, and the NextMarker field will be set. Sending another list_objects with the Marker set to the returned NextMarker will retrieve the next MaxKeys objects. This allows pagination through responses that are consistent even in the face of mutating buckets, in the limited sense that we will get at least all *objects* that were not mutated while paginating.

In order to retrieve a single object, we use get_object:

```
response = boto3.client("s3").get_object(
    Bucket='string',
    Key='string',
)
value = response["Body"].read()
```

The value will be a *bytes* object.

Especially for small- to medium-sized objects, say up to several megabytes, this is a way to allow simple retrieval of all data.

In order to push such objects into the bucket we can use this:

```
response = boto3.client("s3").put_object(
    Bucket=BUCKET,
    Key=some_key,
    Body=b'some content',
)
```

Again, this works well for the case where the body all fits in memory.

As we have alluded to earlier, when uploading or downloading larger files (for example, videos or database dumps) we would like to be able to upload incrementally, without keeping the whole file in memory at once.

The boto3 library exposes a high-level interface to such functionality using the *_fileobj methods.

For example, we can transfer a large video file using:

```
client = boto3.client('s3')
with open("meeting-recording.mp4", "rb") as fpin:
    client.upload_fileobj(
        fpin,
```

```
        my_bucket,
        "meeting-recording.mp4"
    )
```

We can also use similar functionality to download a large video file:

```
client = boto3.client('s3')
with open("meeting-recording.mp4", "wb") as fpout:
    client.upload_fileobj(
        fpin,
        my_bucket,
        "meeting-recording.mp4"
    )
```

Finally, it is often the case that we would like objects to be transferred directly out of S3 or into S3, without the data going through our custom code – but we do not want to allow unauthenticated access.

For example, a continuous integration job might upload its artifacts to S3. We would like to be able to download them through the CI web interface, but having the data pass through the CI server is unpleasant – it means that this server now needs to handle potentially larger files where people would care about transfer speeds.

S3 allows us to generate "pre-signed" URLs. These URLs can be given as links from another web application, or sent via e-mail or any other methods, and allow time-limited access to the S3 resource.

```
url = s3.generate_presigned_url(
    ClientMethod='get_object',
    Params={
        'Bucket': my_bucket,
        'Key': 'meeting-recording.avi'
    }
)
```

This URL can now be sent via e-mail to people who need to view the recording, and they will be able to download the video and watch it. In this case, we saved ourselves any need from running a web server.

An even more interesting use case is allowing pre-signed *uploads*. This is especially interesting because uploading files sometimes requires subtle interplays between the web server and the web application server to allow large requests to be sent in.

Instead, uploading directly from the client to S3 allows us to remove all the intermediaries. For example, this is useful for users who are using some document sharing applications.

```
post = boto3.client("s3").generate_presigned_post(
    Bucket=my_bucket,
    Key='meeting-recording.avi',
)
post_url = post["url"]
post_fields = post["fields"]
```

We can use this URL from code with something like:

```
with open("meeting-recording.avi", "rb"):
    requests.post(post_url,
                  post_fields,
                  files=dict(file=file_contents))
```

This lets us upload the meeting recording locally, even if the meeting recording device does not have S3 access credentials. It is also possible to limit the maximum size of the files via `generate_presigned_post`, to limit the potential harm from an unknown device uploading these files.

Note that pre-signed URLs can be used multiple times. It is possible to make a pre-signed URL only valid for a limited time, to mitigate any risk of potentially mutating the object after uploading. For example, if the duration is one second, we can avoid checking the uploaded object until the second is done.

13.4 Summary

AWS is a popular Infrastructure-as-a-Service platform, which in general is used on a pay-as-you-go basis. It is suitable to automation of infrastructure management tasks, and `boto3`, maintained by AWS itself, is a powerful way to approach this automation.

Index

M. Zadka, *DevOps in Python*, https://doi.org/10.1007/978-1-4842-4433-3

D

E

F

G

H

I

J, K, L